Conception Informatique de Systèmes d'Information

Pôle Informatique 2013-2014

École Nationale Supérieure des Mines
158 cours Fauriel
42023 Saint-Étienne Cedex 02

Antoine Zimmermann

Antoine.Zimmermann@emse.fr
Centre G2I, ENSM-SE
adapté du cours de Laurent Vercouter

Table des matières

1 Introduction sur les Systèmes d'Information **5**
1.1 Notions de base liées au système d'information 5
1.2 Mise en oeuvre du système d'information informatisé 7

2 Le modèle Entité-Association **11**
2.1 Développement et conception d'une base de données 11
2.2 Composantes du modèle EA 13
2.2.1 Entité et type d'entité 14
2.2.2 Attributs et clés 15
2.2.3 Association 16
2.3 Exercices 19

3 Le modèle relationnel **21**
3.1 L'algèbre relationnelle 21
3.1.1 Les bases du modèle relationnel 21
3.1.2 Les opérateurs relationnels de base 24
3.1.3 Combinaison des opérations de base 28
3.1.4 Les opérateurs relationnels complémentaires 28
3.2 Traduction d'un modèle EA vers un modèle relationnel 31
3.2.1 Les règles de transformation 31
3.2.2 Introduction de clés numériques 33
3.3 Dépendances et normalisation 34
3.3.1 Les dépendances fonctionnelles 35
3.3.2 La première forme normale 36
3.3.3 La deuxième forme normale 36
3.3.4 La troisième forme normale 37
3.3.5 La forme normale de Boyce-Codd 38
3.4 Exercices 38

4 Le langage SQL **41**
4.1 Définition de données 41
4.1.1 Création de tables 42
4.1.2 Suppression et modification de tables 43
4.2 Modification de données 44
4.2.1 Insertion de données 44
4.2.2 Suppression de données 45
4.2.3 Mise à jour de données 45
4.3 Interrogation de données 46
4.3.1 Expression de projection 46
4.3.2 Expression de sélection 47
4.3.3 Expression de jointure 48
4.3.4 Interrogations de groupe 48
4.4 Imbrication de requêtes 50
4.4.1 Définition de données par requêtes imbriquées 50
4.4.2 Sélections imbriquées 51

Chapitre 1

Introduction sur les Systèmes d'Information

Ce livret est un support associé au cours "*Conception de Système d'Information*" appartenant au pôle Informatique de la première année du cycle Ingénieur Civil des Mines de l'École Nationale Supérieure des Mines de Saint-Étienne. L'objectif de ce cours est de familiariser les étudiants aux techniques de conception de la composante informatique d'un système d'information, plus particulièrement des bases de données. La modélisation sous la forme d'Entité/Association et suivant le modèle relationnel sont abordés ainsi que l'implémentation informatique des modèles obtenus et sa manipulation à l'aide du langage SQL. Cette introduction commence par expliciter ce que recouvre le système d'information en identifiant plus précisément les notions qui relèvent de l'informatique.

1.1 Notions de base liées au système d'information

Selon Le Moigne [1], le système d'information (SI) est vu comme un sous-système de l'entreprise lié au système opérant et au système de pilotage (cf. figure 1.1). Le système opérant réalise de nombreuses opérations élémentaires dépendantes du métier de l'entreprise (par exemple la transformation de matières prémières en produit dans une usine, ou des mouvements financiers dans une banque) qui requièrent des informations (le détail d'une commande, l'état d'un stock, le temps de travail du personnel, etc) et qui en génèrent (état de la production, établissement de factures, etc). De l'autre côté, le système

1. *La modélisation des systèmes complexes*, J.-L. Le Moigne, Dunod, 1990

de pilotage s'appuie sur le système d'information pour prendre des décisions stratégiques qui génèrent également de nouvelles informations. Ainsi le système d'information de l'entreprise est à la fois une ressource utilisée aux niveaux stratégiques et opérationnelles mais aussi un moyen d'agir sur le fonctionnement de l'entreprise.

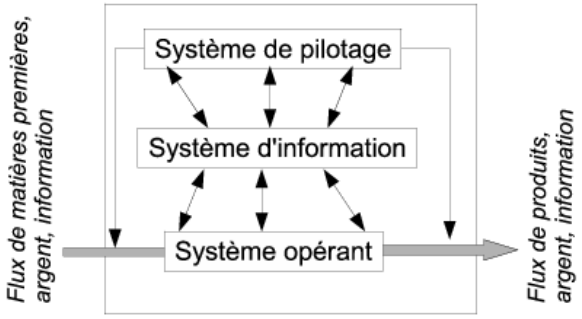


FIGURE 1.1 – Le système d'information dans l'entreprise

Le système d'information remplit deux fonctions principales :

- La **production d'information** qui recouvre les sous-fonctions suivantes :
  - *Collecter des informations* : le SI doit être capable d'accéder à des informations circulant dans l'entreprise.
  - *Traiter et transmettre des informations* : les informations collectées doivent pouvoir être interprétées, subir des traitements pour éventuellement produire de nouvelles informations et être communiquées de manière pertinente à des entités de l'entreprise (personnels, services, logiciels, etc).
  - *Mémoriser des informations* : le SI doit stocker de manière pérenne les informations.
- La **mise en œuvre d'outils de gestion** selon différentes dimensions :
  - la *dimension technologique* reprsente l'ensemble des outils nécessaires à la production d'information. Cela recouvre le matériel à la fois informatique (ordinateur, serveur, imprimantes, etc) et non informatique (documents papiers, tableaux d'affichage, etc), les logiciels, les méthodes de travail, le savoir-faire, etc.
  - la *dimension économique* tablit le coût de mise en place du système d'information ;

- la *dimension sociale* car le système d'information est utilisé et concerne des acteurs humains qu'il faut prendre en compte à la fois pour leur permettre d'utiliser le SI (adaptation du SI aux besoins et savoir-faire des acteurs) et dans les informations manipulées (confidentialité, respect de la vie privée, etc).

Enfin, le système d'information doit être vu comme beaucoup plus large que sa seule partie informatique. Le **Système d'Information Opérationnel** (SIO) représente l'ensemble des informations circulant dans l'entreprise que ce soit par le biais de technologies informatiques mais aussi par des moyens de communication classiques tels que les supports papiers pour communiquer des notes de services, des ordres de fabrication, etc, des communications téléphoniques ou même des discussions entre employés lors des pauses. Le SIO est un système qui peut être vu comme "naturel et vivant" qui n'est pas consciemment construit et qui est présent de manière diffuse dans toute l'entreprise. Le **Système d'Information Informatisé** (SII) est un sous-ensemble du SIO. Celui-ci est artificiel et construit dans l'objectif de faciliter le fonctionnement du SIO et de pouvoir influer dessus dans un objectif d'efficacité de fonctionnement de l'entreprise. La suite de ce cours se focalise exclusivement sur le SII.

1.2 Mise en oeuvre du système d'information informatisé

Le Système d'Information Informatisé repose sur le stockage de données numérisées représentant les informations mémorisées par le SI. Ces données sont regroupées dans des fichiers dont l'accès est géré par un Système de Gestion de Fichiers (SGF). Ces fichiers sont organisés selon une structure très simple telle qu'une arborescence de dossiers. La structure d'un SGF est alors insuffisante pour organiser efficacement les données à stocker et il est nécessaire d'y ajouter une *base de données* qui définit une structure plus riche pour représenter les informations.

Une **base de données** (BDD) est un ensemble d'informations accessible par un ordinateur selon une structure définie pour satisfaire simultanément plusieurs utilisateurs. Il est primordial de considérer l'objectif pour lequel est conçu la base de données pour représenter les informations pertinentes à son utilisation. Il existe différentes familles de BDD qui diffèrent par l'approche suivie pour concevoir la structure des données :

- l'approche hiérarchique (proposée vers les années 1960) qui organise les données hiérarchiquement selon une arborescence ;

- l'approche relationnelle (proposée vers les années 1970) qui établit des relations entre des blocs d'information ;
- l'approche objet (proposée vers les années 1990) qui regroupe en blocs des types d'information associés à leurs traitements ;
- l'approche NoSQL (proposée vers les années 2000) qui se décompose en diverses approches s'éloignant de l'omniprsent modèle relationnel : approche clé-valeur, approche document, approche graphe.

L'approche relationnelle est actuellement l'approche dominante car la plus utilisée dans les solutions commerciales existantes. C'est celle que nous adoptons dans ce cours et qui est détaillée dans le chapitre 3.

Le **Système de Gestion de Bases de Données** (SGBD) est un ensemble d'outils logiciels permettant la manipulation de BDD. L'objectif de ces outils est non seulement de permettre aux concepteurs et administrateur de la BDD de la construire et de la maintenir mais aussi de permettre sa manipulation (consultation de données, modifications, etc) par des utilisateurs non-informaticiens.

Il est attendu d'un SGBD qu'il respecte les principes fondamentaux qui suivent.

- **Fidélité** : la BDD doit fournir une image fidèle de la réalité qu'elle modélise.
- **Unicité** : la BDD ne doit pas contenir d'informations redondantes.
- **Indépendance** : le modèle de stockage ne doit pas avoir d'influence sur le SGBD, pour favoriser les évolutions et améliorations.
- **Concurrence** : le SGBD doit pouvoir gérer plusieurs accès simultanés à une même donnée.
- **Performance** : les temps d'exécution doivent être raisonnables.
- **Confidentialité** : l'accessibilité des données dépend de l'utilisateur, et toutes les données ne sont pas forcément accessibles à tous les utilisateurs.
- **Intégrité** : le SGBD doit fournir des garanties de fiabilité et de cohérence.
- **Robustesse** : le SGBD doit être tolérant aux problèmes matériels, logiciels ou humains.

Une BDD peut être considérée à différents niveaux selon que l'on se préoccupe de sa conception, de son utilisation ou du détail du stockage physique des données.

- Le niveau **externe** s'intéresse à l'utilisation de la BDD par les utilisateurs finaux. On y définit les interfaces de consultation et de modification des utilisateurs. Ce niveau est géré conjointement par les utilisateurs et les concepteurs de la BDD.
- Le niveau **conceptuel** identifie les informations à représenter et dresse

un modèle de la réalité observée. Il est géré par le concepteur de la BDD.

- Le niveau **logique** formalise le modèle conceptuel en utilisant un langage de définition des données propre à une famille de BDD. Il est géré par le concepteur de la BDD.
- Le niveau **physique** s'intéresse aux stockages des données sur des supports numériques. Il est en général géré automatiquement par le SGBD avec quelques interventions du concepteur.

Dans la suite de ce cours, nous détaillons la conception des niveaux conceptuels et logiques d'une BDD. Dans le chapitre 2, nous nous intéressons au niveau conceptuel. Le chapitre 3 est consacré au niveau logique ainsi que le chapitre 4 qui présente le langage SQL permettant l'implantation et la manipulation d'une BDD relationnelle.

Chapitre 2

Le modèle Entité-Association

2.1 Développement et conception d'une base de données

La réalisation d'une base de données, support de stockage et d'accès aux données du SI, nécessite l'adoption d'une méthodologie précise dictant les étapes à suivre. A l'instar du développement de toute application informatique conséquente, il convient de suivre un cycle de développement. Différents visions d'un cycle de développement existent. La figure 2.1 illustre la vision la plus classique car la plus ancienne qui est un cycle de développement linéaire.

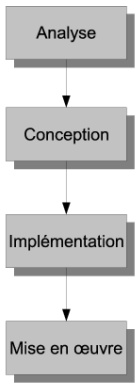


FIGURE 2.1 – Développement linéaire

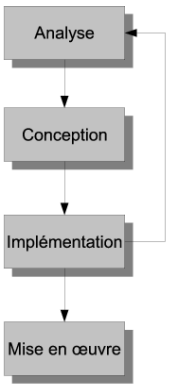


FIGURE 2.2 – Développement cyclique

Le développement est ici décomposé en quatres étapes :

1. l'**analyse** a pour objectif de définir les fonctions attendues du système. Le client de l'application est fortement impliqué dans cette phase notamment dans l'expression de ses besoins. La faisabilité et le coût de ces besoins sont étudiés et peuvent donner lieu à une négociation entre le client et les développeurs de l'application. À l'issue de cette phase, le périmètre du développement attendu est explicité dans le souci de supprimer toute ambiguïté et d'éviter d'éventuels désaccords entre client et développeur à la livraison de l'application. Du point de vue du développement de l'application, l'objectif de cette phase est de fournir une description fonctionnelle détaillant les opérations attendues, les flux (entrées/sorties) d'information et de contrôle entre utilisateurs et machines, les opérations manuelles nécessaires, etc.
2. la **conception** est l'étape de création d'un modèle de l'application attendue. A partir de la description fonctionnelle obtenue lors de l'analyse, l'architecture globale de l'application est définie en utilisant un langage formel spécifique au type de développement attendu. Lors de la conception d'une base de données par exemple, il s'agit de définir la structure des informations manipulées : quelles sont les données élémentaires ? Quel est leur type ? Lesquelles doivent être regroupées ? Quelles relations entre elles ? Les modèles étudiés dans la suite de ce cours fournissent les éléments permettant de construire un modèle de conception.
3. l'**implémentation** passe du modèle conceptuel précédemment obtenu à un logiciel satisfaisant les besoins émis par le client. Plus les phases d'analyse et de conception sont minutieuses, plus l'implémentation est simplifiée. Celle-ci pouvant aller jusqu'à une simple "traduction" depuis le langage formel utilisé pour la conception vers un langage informatique. Il est possible qu'à cette étape, on se rende compte d'un décalage entre les besoins du client et le résultat obtenu (par exemple une mauvaise estimation de la faisabilité de certaines fonctions ou une conception inadéquate). Une alternative au cycle de développement linéaire, le développement cyclique (cf figure 2.2) prévoit la possibilité de retour vers l'étape d'analyse pour revoir le cahier des charges ou le modèle conceptuel.
4. la **mise en œuvre** contient le déploiement de l'application sur l'infrastructure matérielle et logicielle du client. Les problèmes liés à la prise en compte de son environnement informatique spécifique sont ici traités. La mise en place d'une nouvelle organisation autour de ce nouvel outil est aussi concernée, ainsi que des opérations de formation des utilisateurs, de maintenance et de support du logiciel.

Plusieurs alternatives à ces visions classiques du cycle de développement ont été proposées ces dernières décennies, privilégiant notamment le niveau de décomposition entre phases et les allers-retours entre celles-ci. Par exemple, la figure 2.3 illustre le cycle en V décomposant plus finement les étapes et prévoyant des retours horizontaux entre les étapes situées sur la même ligne. Le but de ces nouvelles propositions de cycle est d'éviter l'effet "tunnel" qui consiste à considérer les besoins du client au début du cycle, de dérouler toutes les étapes méthodiquement jusqu'à la fin du développement où l'on revient sur l'adéquation du logiciel produit avec les besoins initiaux. L'inconvénient majeur de cet effet tunnel est qu'en cas de problème, tout doit être recommencé depuis le début provoquant un coût considérable.

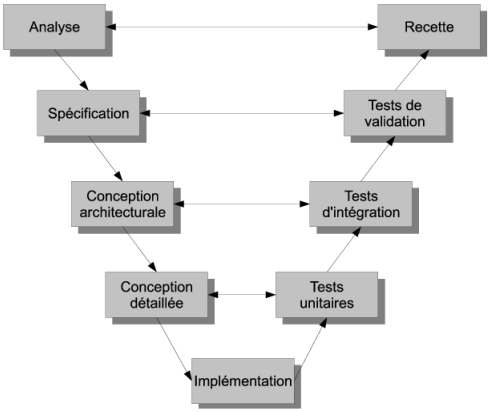


FIGURE 2.3 – Cycle en V

Nous ne détaillons pas ici tous ces cycles de développement et leurs caractéristiques. Il faut cependant avoir conscience de l'existence de différentes étapes lors du développement d'une application informatique, a fortiori une base de données, qui ne se limite pas à une simple implémentation dans un langage informatique. La phase de conception est primordiale, utilisant un langage de modélisation tel que le modèle EA décrit dans la section suivante.

2.2 Composantes du modèle EA

Le modèle EA (Entité-Association) a été proposé pour la première fois en 1977 par P. Chen pour la conception de modèles d'organisation de données. Ce modèle fournit un petit nombre de concepts simples visant à représenter de manière abstraite une réalité observée. L'objectif est d'obtenir ce que l'on appelle un **Modèle Conceptuel de Données** (MCD). Il est composé

des concepts d'*entité* et *types d'entité*, d'*attribut*, de *clé* et d'*association*. La terminologie associée à ces concepts n'est pas toujours unique et le terme "relation" est parfois utilisé à la place d'"association", celui de "propriété" à la place d'"attribut", celui de "classe d'entité" à la place de "type d'entité et celui d'"identifiant" à la place de "clé". Parfois même, on utilise le terme "entité" pour désigner des types d'entités et pour distinguer les entités de leur type, on utilise le terme d'"occurrence".

2.2.1 Entité et type d'entité

Une **entité** est un objet abstrait ou concret provenant de l'observation du monde réel et pour lequel nous souhaitons enregistrer et connaître des informations. Un **type d'entités** identifie un ensemble d'entités ayant des caractéristiques communes. La difficulté de la conception revient à identifier les bons types d'entités, sachant que plusieurs possibilités sont souvent envisageables et qu'une "bonne" conception dépend fortement de l'utilisation prévue des données.

Pour donner des exemples du modèle EA, nous allons considérer un cas simple de gestion de cartes grises. Dans cet exemple, on pourrait avoir un type d'entités appelé `Véhicule` représentant les informations associées à une voiture et un type d'entités appelé `Propriétaire` pour décrire les personnes propriétaires d'un véhicule.

Graphiquement on représente un type d'entité par un rectangle découpé horizontalement en deux parties et dont la partie supérieure contient le nom du type d'entités. Notre exemple est représenté par la figure 2.4.

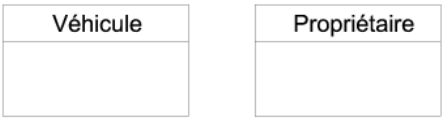


FIGURE 2.4 – Exemples de type d'entités

On peut remarquer sur cet exemple que l'utilisation attendue des entités a déjà un impact sur la description des entités. On souhaite en effet représenter un individu par son lien de propriété sur des véhicules ce qui implique le choix de `Propriétaire` comme nom d'entité. Dans d'autres applications le même individu pourrait être représenté par une entité de type `Personne`, `Client` ou `Patient`, par exemple.

Puisque ce sont les types d'entités, et non les entités, qui intéressent les concepteurs de SI, il arrive qu'on utilise le terme "entité" à la place de

"types d'entités". Pour lever l'ambiguité lorsque l'on veut se réferrer à une entité seule (une instance de type), on utilise le terme **occurrence** d'une entité. Elle représente un objet spécifique du type d'entités. Par exemple, la Peugeot 206 immatriculée "1234 WW 42" est une entité du type `Véhicule`, et la personne prénommée "Paul Martin" née le 4 février 1980 est une entité de type `Propriétaire`.

2.2.2 Attributs et clés

Un **attribut** est une caractéristique d'une entité (ou d'une association) que le concepteur juge nécessaire de répertorier mais qu'il n'est pas ncessaire de dcomposer plus prcisment. Un attribut est associé à un type d'entités (ou à une association) qui précise son type (chaîne de caractères, entier, date, etc). Graphiquement les attributs sont représentés dans la partie inférieure d'un type d'entités.

Par exemple, le type `Véhicule` peut disposer de deux attributs `immatriculation` et `marque` qui sont deux chaînes de caractères, et le type `Propriétaire` des attributs `nom` et `prénom`, deux chaînes de caractères, et `naissance`, une date (cf. figure 2.5). Pour chaque attribut, on précise généralement le type de données correspondant, même si le modèle EA permet d'exprimer une information incomplte. Les types de données utilisés pour les attributs sont habituellement les types numériques (entiers, flottants), les chaâines de caractères et les dates et heures. Il est néanmoins possible d'introduire d'autres types de données dans certains cas.

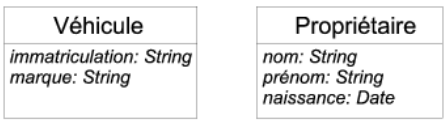


FIGURE 2.5 – Exemples d'attributs

Une entité attribue une valeur spécifique à chaque attribut. Pour les deux occurrences citées ci-dessus en exemple, l'occurence de `Véhicule` donnerait la valeur "Peugeot 206" à l'attribut `marque` et "1234 WW 42" à l'attribut `immatriculation` et l'occurrence de `Propriétaire` donnerait la valeur "Martin" à l'attribut `nom`, "Paul" à l'attribut `prénom` et "4 février 1980" à l'attribut `naissance`.

Une **clé** est un attribut ou un ensemble d'attributs qui permet d'identifier de manière unique une occurrence d'un type d'entités parmi toutes

ses occurrences. Graphiquement, on souligne les attributs composant la clé d'une entité. Dans le type `Véhicule`, la clé est l'attribut `immatriculation` car deux véhicules distincts ne peuvent avoir la même immatriculation. Le choix de la clé est moins simple pour le type d'entités `Propriétaire` car il est possible que plusieurs personnes aient le même nom, le même prénom ou la même date de naissance. On peut dans ce cas opter pour une clé composée de l'ensemble de ces trois attributs. Cela signifie qu'il ne peut pas y avoir dans notre modèle, deux propriétaires nés le même jour et ayant les mêmes nom et prénom. La figure 2.6 représente cet exemple.

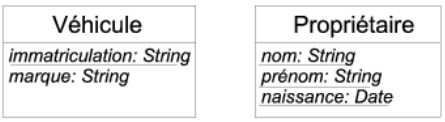


FIGURE 2.6 – Exemples de clés

La présence d'une clé pour un type d'entités n'est pas obligatoire mais il est généralement conseillé de la fournir afin de savoir comment l'on compte identifier les entités.

2.2.3 Association

Une **association** représente un lien entre des entités présentant un intérêt pour la conception que l'on souhaite réaliser. De la même manière que l'on s'intéresse aux *types* d'entités pour décrire le SI, on s'intéresse aux **types d'associations**. Un type d'associations n'a d'existence que relativement aux types d'entités qu'elle relie. On utilise souvent un verbe pour représenter un type d'associations. Graphiquement, un type d'association est représentée par un ovale découpé horizontalement en deux parties et dont la partie supérieure contient le nom de l'association. Comme pour les types d'entités, on se contente parfois d'appeler "association" un type d'associations et nous nous permettons cet abus par la suite car il est rarement gênant.

Dans notre exemple, l'association `Possède` relie les types d'entités `Véhicule` et `Propriétaire` tel que représenté dans la figure 2.7.

Une association peut relier plus de deux entités et peut également contenir des attributs. La figure 2.8 en donne un exemple. Ici, l'association `Achète` associe un propriétaire (l'acheteur), un concessionnaire (le vendeur) et un véhicule (l'objet acheté). L'attribut `dateAchat` attribué à l'association indique quand l'achat a été effectué.

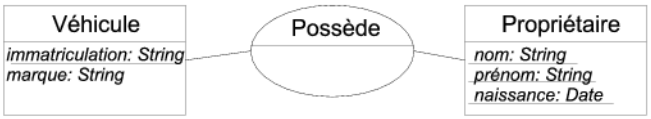


FIGURE 2.7 – Exemple d'association binaire

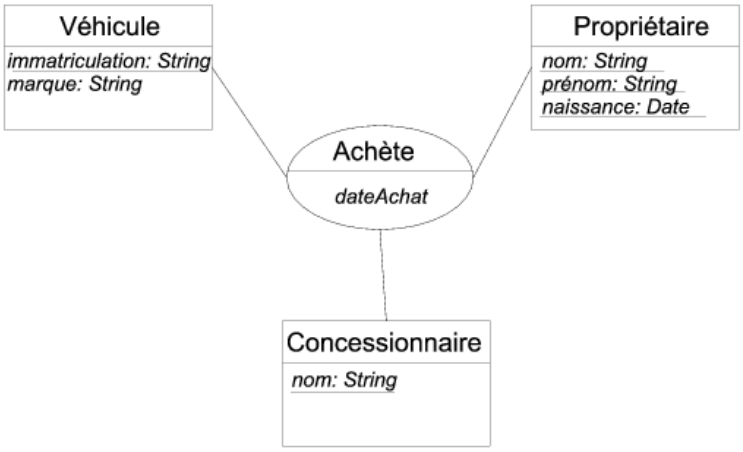


FIGURE 2.8 – Exemple d'association ternaire avec attribut

La **cardinalité** d'une association indique le nombre d'occurrences de chaque entité qui peuvent être impliquées dans une même association. La cardinalité d'une entité figure sur le trait reliant l'entité à l'association et est composée de deux nombres séparés par '`:`', les nombres minimum et maximum d'occurrences impliqués dans l'association. Le symbole N est utilisé pour indiquer qu'un nombre illimité d'occurrences d'un même type d'entités peut participer à une même association.

Les cardinalités doivent se lire ainsi : on considère la cardinalité x:y se trouvant à proximité d'un type d'entités T_1, pour l'association A, qui relie T_1 à T_2. Alors cela signifie qu'un entité de type T_1 est en relation par A avec *au minimum* x entités de type T_2, et *au maximum* y entités de type T_2. Si y est le symbol N alors il n'y a pas de maximum, c'est-à-dire que N peut être entendu comme le symbole ∞.

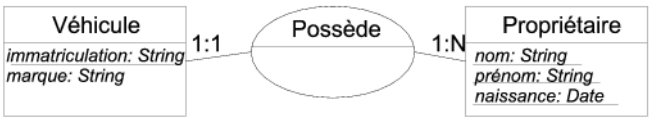


FIGURE 2.9 – Exemple de cardinalités

La figure 2.9 reprend l'exemple précédent en y ajoutant des cardinalités. On y remarque qu'un propriétaire peut être associé dans une association `Possède` à plusieurs véhicules (mais au minimum à 1 véhicule). De l'autre côté, un véhicule est associé par l'association `Possède` à un et un seul propriétaire.

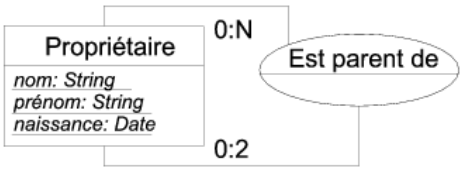


FIGURE 2.10 – Exemple d'entité reliée à elle-même

Enfin, on peut noter qu'il est possible qu'un type d'entités ait une association avec elle-même. La figure 2.10 en donne un exemple. L'association `Est parent de` relie des occurrences du même type `Propriétaire`. Les cardinalités signifient qu'une occurrence de `Propriétaire` peut être parente de

0 à plusieurs occurrences de `Propriétaire`, et que de l'autre côté une occurrence de `Propriétaire` a de 0 à 2 parents (évidemment, une personne a nécessairement des parents, mais pour les besoins du système d'information, on peut autoriser une personne à n'avoir aucun parent reconnu).

La lecture des cardinalités pour une association reliant trois ou plus types d'entités est plus délicate. On lit alors une cardinalité x:y proche du type d'entités T pour le type d'associations A comme suit : une entité de type T entre dans au moins x associations de type A et au plus y associations de type A.

2.3 Exercices

Vous devez concevoir un système d'information pour gérer la vente de billets pour des spectacles. Voici un descriptif simplifié des besoins du maître d'ouvrage : *Notre bureau de vente virtuel en ligne vend des billets pour des spectacles, des représentations musicales et des concerts. Un spectacle a lieu à un endroit donné, à une date et à une heure spécifique. Le lieu du spectacle a une certaine capacité (nombre de place assises ou debout selon les cas). Ce nombre correspond au nombre maximum de billets disponibles pour le spectacle. Chaque spectacle a un artiste principal et éventuellement un ou plusieurs artistes additionnels (notamment les "premières parties" de concerts). Les artistes ont une nationalité et correspondent à un genre. Un spectacle a un prix. Quand les clients achètent des billets, leur commande est enregistrée avec la date de commande, le nom du client, et le nombre de billets achetés. Un client peut annuler une commande mais seulement avant les 24h précédant le spectacle.*

Exercice 1 : *Réaliser un modèle Entité-Association de cette application.*

Après avoir consulté votre proposition de modèle, le maître d'ouvrage s'aperçoit que cela ne suffit pas pour ses besoins. Il voudrait que les genres des artistes soient organisés en hiérarchie (par exemple, le punk est un sous-genre du rock) et veut établir un programme de fidélité pour ses clients. Chaque commande est alors associée à un client enregistré avec son nom et son adresse et un indicateur spécifiant si le client possède une carte de fidélité.

Exercice 2 : *Modifiez votre modèle Entité-Association pour tenir compte de ce besoin supplémentaire.*

Chapitre 3

Le modèle relationnel

Le passage d'un modèle EA à une implantation informatique n'est pas immédiat. Par exemple, une association de dimension supérieure à 2 doit être transformée ou des associations ayant des attributs ne sont pas toujours implémentables en l'état. Le modèle conceptuel de données décrit par un modèle EA reste indépendant de tout choix technologique et doit être transformé en un **Modèle Logique de Données** (MLD) avant d'être implémenté. Différents types de MLD existent selon l'approche de représentation (objets, hiérarchique, etc). Nous détaillons ici le modèle relationnel.

3.1 L'algèbre relationnelle

Le modèle relationnel de données a été proposé en 1970 par un mathématicien américain E. F. Codd qui a appliqué la "théorie des relations" à la gestion de fichiers informatiques. Le modèle relationnel a été industrialisé pendant les années 1980 dans plusieurs SGBD relationnels, dont les plus connus sont Oracle, DB2, Sybase, Ingres, Postgres, MySQL, qui dominent le marché. Le modèle relationnel repose sur une algèbre relationnelle dotée de concepts simples dont la combinaison fournit une base puissante et solide.

3.1.1 Les bases du modèle relationnel

Suivant le modèle relationnel, les données sont structurées en **relations**. Une relation est un espace en deux dimensions où sont stockées les données. Une de ces dimensions représente les attributs qui prennent leur valeur dans un domaine.

Définitions préliminaires

Un **attribut** A_i est une variable qui prend ses valeurs dans un **domaine** D_i. Soit U un ensemble d'attributs tel que $U = \{A_1, A_2, ..., A_n\}$. Une **relation** r définie sur U est un sous-ensemble du produit cartésien $D_1 \times D_2 \times ... \times D_n$. Un **schéma de relation** définit la structure de la relation, c'est-à-dire le nom des attributs associé à leur domaine. Le schéma de relation de r s'écrit $R(A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$. Une **clé primaire** pour U est un sous-ensemble des attributs de U. Les attributs constituant la clé de r sont écrits en premier dans le schéma de relation et sont soulignés. Une **base de données relationnelle** est un ensemble de relations $r_1, r_2, ...r_n$.

Exemple

Le domaine de valeurs de l'attribut `marque` de `Vehicule` est : {Peugeot ; Renault ; BMW}. Celui de l'attribut `immatriculation` est une chaîne de caractères composée de 4 chiffres suivi de 2 lettres puis de 2 chiffres, soit { '0001 AA 01', '0002 AA 02', ... '9999 ZZ 99'}. Le produit cartésien des domaines de valeurs est représenté dans la table 3.1.

Peugeot	'0001 AA 01'
Renault	'0001 AA 01'
BMW	'0001 AA 01'
Peugeot	'0001 AA 02'
...	...
BMW	'9998 ZZ 99'
Peugeot	'9999 ZZ 99'
Renault	'9999 ZZ 99'
BMW	'9999 ZZ 99'

TABLE 3.1 – Produit cartésien de deux domaines

Si la relation `Véhicule` est composée de ces deux attributs, elle contient un sous-ensemble de ce produit cartésien.

Définitions complémentaires

Une relation peut également être appelée *table* puisque l'on peut toujours représenter son contenu sous forme de table, comme en témoigne la table 3.1. Un attribut d'une relation est alors également appelée *colonne*. Une *ligne* d'une relation est appelée un **tuple**. Une relation est donc constituée de plusieurs tuples parfois appelés n-uplet . Soit $u \in r$ un tuple de la relation r,

on note $u.A_i$ la valeur de l'attribut A_i du tuple u. Chaque tuple est distinct, c'est-à-dire que $\forall u, v \in r$, si $\forall i \in \{1, 2, ..., n\}, u.A_i = v.A_i$, alors $u = v$.

Enfin, une clé de la relation r est un ensemble minimal d'attributs $K \subset \{A_1, A_2, ..., A_n\}$ tels que les valeurs de ces attributs permettent de distinguer tout tuple des autres.

$\forall u, v \in r,\ si\ u.K = v.K,\ alors\ u = v$

Il se peut que plusieurs possibilités existent pour le choix de la clé d'une relation. Ces possibilités sont appelées des *clés candidates*. Il faut cependant choisir pour chaque relation une unique clé qui sera appelée la **clé primaire**.

Les clés d'une relation sont utilisées pour créer des associations entre relations. Pour que les tuples d'une relation r puissent faire référence à des tuples de la relation r', les attributs de la clé primaire de r' doivent être ajoutés aux attributs de la relation r. Un tuple $u \in r$ peut alors désigner de manière unique un tuple $v \in r'$ en assignant les valeurs de $v.K$ aux attributs correspondant de u.

Exemple

Soit la relation `Proprietaire` correspondant au schéma :

`Proprietaire`(<u>`numero:int`</u>, `nom:string`, `prenom:string`, `naissance:date`).

La table 3.2 donne des exemples de tuples appartenant à cette relation.

<u>numero</u>	nom	prenom	naissance
5	Martin	Paul	4 février 1980
13	Dupont	Marcel	15 mai 1976
76	Durand	Pierre	29 novembre 1978

TABLE 3.2 – tuples de la relation Propriétaire

La relation `Vehicule` utilise la clé primaire de `Proprietaire` pour qu'un véhicule puisse être relié à son propriétaire. Son schéma est :

`Vehicule`(<u>`immatriculation:string`</u>, `marque`:{Peugeot ; Renault ; BMW}, `proprietaire:int`).

Des exemples de tuples sont donnés table 3.3.

Clé étrangère

Dans un schéma relationnel, une clé étrangère est un attribut ou un ensemble d'attributs d'une relation dont les valeurs se réfèrent à une clé primaire d'une autre relation. Par exemple, dans l'exemple qui précèdent, la

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

TABLE 3.3 – tuples de la relation `Véhicule`

relation `Vehicule` possède une clé étrangère `proprietaire` qui se réfère à la clé primaire `numero` de la relation `Proprietaire`.

Il n'y a pas de notation standardisée pour indiquer qu'un attribut est une clé étrangère. Cependant, on peut le montrer à l'aide d'une flèche allant de `proprietaire` à `numero` ou bien en le précisant en français à la suite du schéma des relations :

`Proprietaire`(`numero:int`, `nom:string`, `prenom:string`, `naissance:date`).
`Vehicule`(`immatriculation:string`, `marque:`{Peugeot ; Renault ; BMW}, `proprietaire:int`).

L'attribut `proprietaire` dans la relation `Vehicule` se réfère à `numero` dans la relation `Proprietaire`.

3.1.2 Les opérateurs relationnels de base

La notion de relation permet de définir la structure des données que l'on souhaite stocker. Afin de gérer l'information, il faut pouvoir isoler une portion des données sur lesquelles on effectuera des traitements. Par exemple, si l'on a une relation `Client` contenant 10000 clients, on peut vouloir obtenir les données concernant un client spécifique dont on connait le nom. Ou bien, on peut ne vouloir connatre que les prénoms des clients habitant une certaine région.

Remarque : les annexes de ce livret contiennent une copie de diapositives comprenant des exemples supplémentaires.

L'algèbre relationnelle définit un ensemble d'opérateurs de manipulation des données, plus particulièrement pour interroger une base de données. Les opérateurs unaires s'appliquent sur une unique relation et sont :

- la projection
- la sélection
- *le complément*

Les opérateurs binaires impliquent deux relations et sont :

- l'union
- la différence
- le produit cartésien
- *l'intersection*
- *la division*
- *le thêta-produit*
- *les jointures*

Les opérateurs indiqués en texte simple sont les cinq opérateurs de base de l'algèbre relationnelle. Les opérateurs indiqués en italique sont des opérateurs secondaires qui peuvent être obtenus par combinaison des opérateurs de base.

La projection

La projection est une opération qui consiste à ne considérer qu'une partie des attributs d'une relation. La projection d'une relation R sur un sous-ensemble de ses attributs $\{A_{i_1}, A_{i_2}, ..., A_{i_k}\} \in \{A_1, A_2, ..., A_n\}$ s'écrit :

$$\prod_{A_{i_1}, A_{i_2}, ..., A_{i_k}} (R)$$

La figure 3.1 montre un exemple de projection de la relation `Vehicule` sur les attributs `immatriculation` et `proprietaire`, en utilisant la notation graphique appropriée.

FIGURE 3.1 – Exemple de projection

La sélection

La sélection (également appelée restriction) est une opération qui consiste à ne considérer que les tuples d'une relation qui satisfont certaines conditions. Une sélection sur une relation R selon un ensemble de conditions C se note :

$$\sigma_C(R)$$

La figure 3.2 montre un exemple de selection sur la relation `Vehicule` avec la condition que le propriétaire soit le numéro 76, en utilisant la notation graphique appropriée.

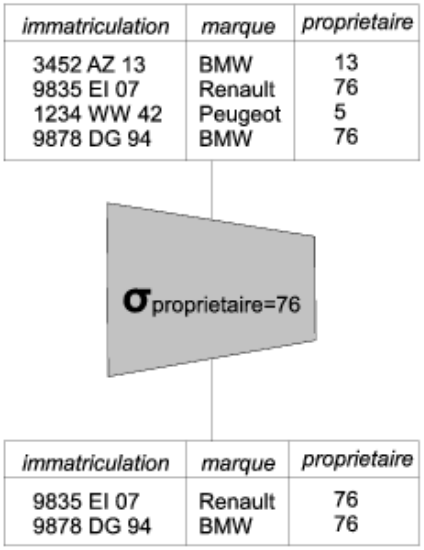


FIGURE 3.2 – Exemple de sélection

L'union

L'union est une opération binaire qui produit une relation, fusion de deux relations. Ceci n'est possible que sur deux relations de même schéma. C'est une opération commutative. Les tuples en double (présents dans les deux relations) sont éliminés. L'union de deux relations R et R' s'écrit :

$$R \cup R'$$

La figure 3.3 montre un exemple d'union de deux relations R et R'.

La différence

La différence est une opération binaire qui permet d'obtenir les tuples d'une relation qui n'apparaissent pas dans une seconde relation. Ici encore,

R

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

R'

immatriculation	marque	proprietaire
3976 RS 43	BMW	5
9835 EI 07	Renault	76

R U R'

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76
3976 RS 43	BMW	5

FIGURE 3.3 – Exemple d'union

cette opération n'est possible que sur deux relations de même schéma. La différence de deux relations R et R' s'écrit :

$$R - R'$$

La figure 3.4 montre un exemple de différence entre deux relations R et R'.

R

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

R'

immatriculation	marque	proprietaire
3976 RS 43	BMW	5
9835 EI 07	Renault	76

R - R'

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

FIGURE 3.4 – Exemple de différence

Le produit cartésien

Le produit cartésien est une opération binaire produisant une relation contenant toutes les possibilités de combinaison entre les tuples des deux relations. Le produit cartésien est une opération coûteuse qui est rarement utilisée seule. Le produit cartésien entre deux relations R et R' s'écrit :

$$R \times R'$$

La figure 3.5 montre un exemple de produit cartésien entre deux relations R et R'.

R

immatriculation	marque	proprietaire
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

R'

numero	nom	prenom	naissance
5	Martin	Paul	4 février 1980
13	Dupont	Marcel	15 mai 1976
76	Durand	Pierre	29 novembre 1978

R x R'

immatriculation	marque	proprietaire	numero	nom	prenom	naissance
3452 AZ 13	BMW	13	5	Martin	Paul	4 février 1980
3452 AZ 13	BMW	13	13	Dupont	Marcel	15 mai 1976
3452 AZ 13	BMW	13	76	Durand	Pierre	29 novembre 1978
9835 EI 07	Renault	76	5	Martin	Paul	4 février 1980
9835 EI 07	Renault	76	13	Dupont	Marcel	15 mai 1976
9835 EI 07	Renault	76	76	Durand	Pierre	29 novembre 1978
1234 WW 42	Peugeot	5	5	Martin	Paul	4 février 1980
1234 WW 42	Peugeot	5	13	Dupont	Marcel	15 mai 1976
1234 WW 42	Peugeot	5	76	Durand	Pierre	29 novembre 1978
9878 DG 94	BMW	76	5	Martin	Paul	4 février 1980
9878 DG 94	BMW	76	13	Dupont	Marcel	15 mai 1976
9878 DG 94	BMW	76	76	Durand	Pierre	29 novembre 1978

FIGURE 3.5 – Exemple de produit cartésien

3.1.3 Combinaison des opérations de base

Ces opérations de base sont rarement utilisées seules. La puissance de l'algèbre relationnelle s'exprime dans leur combinaison permettant d'exprimer des requêtes précises. Par exemple, une requête retrouvant l'immatriculation des véhicules possédés par la personne dont le nom est "Durand" peut s'écrire :

$$\prod_{\texttt{immatriculation}} (\sigma_{\texttt{proprietaire=numero}\}}(\texttt{Vehicule} \times \sigma_{\texttt{nom="Durand"}}(\texttt{Proprietaire})))$$

Cette formule réalise les opérations suivantes :

1. une sélection sur la relation `Proprietaire` pour ne garder que le(s) tuple(s) dont l'attribut `nom` vaut `"Durand"` ;
2. un produit cartésien du résultat avec la relation `Vehicule` ;
3. une sélection du résultat de ce produit pour ne garder que les tuples où l'attribut `proprietaire` a la même valeur que l'attribut `numero` ;
4. une projection pour ne garder que l'attribut `immatriculation` du résultat.

3.1.4 Les opérateurs relationnels complémentaires

Une combinaison classique de certains opérateurs de base permet d'obtenir des opérateurs dits complémentaires, fréquemment utilisés.

Le complément

Le complément est l'opération qui consiste à construire tous les tuples qui n'existent pas dans une relation. Le complément d'une relation R se note :

$$-R$$

Le complément n'est presque jamais utilisé seul. En effet, le complément d'une relation est souvent infini car il peut contenir, par exemple, l'ensemble de tous les nombres entier ne figurant pas dans une relation.

L'intersection

L'intersection est une opération binaire qui créé une relation ne contenant que les tuples présents dans deux relations. Ceci n'est possible que sur deux relations de même schéma. C'est une opération commutative. L'intersection de deux relations R et R' s'écrit :

$$R \cap R'$$

En supposant que l'on dispose d'une relation `Proprietaire` et d'une relation `Conducteur`, on peut souhaiter connaître les propriétaires qui sont également conducteur (certains conducteurs ne conduisent pas leur propre voiture et certains propriétaires peuvent avoir un chauffeur).

La division

La division est une opération binaire qui génère une relation regroupant exclusivement toutes les parties de tuples d'une relation qui sont associées à tous les tuples d'une autre relation. La division d'une relation R par une relation R' s'écrit :

$$R \div R'$$

Le Thêta-produit

Le Thêta-produit est un produit cartésien de deux relations suivi d'une sélection. Il consiste à ne sélectionner dans le résultat du produit que les tuples qui satisfont une certaines conditions. Le Thêta-produit des relations R et R' selon une condition C se note :

$$R \ \Theta_C \ R'$$

Le Thêta-produit se déduit de la sélection et du produit cartésien comme suit : $R\ \Theta_C\ R' \ \equiv\ \sigma_C(R \times R')$.

Les jointures

La **jointure naturelle** est l'opération la plus utilisée avec la sélection et la projection. C'est un cas particulier de Thêta-produit où la condition représente une égalité entre des attributs des deux relations concernées. La jointure naturelle des relations R et R' selon une condition C se note :

$$R \ \bowtie_C \ R'$$

Comme pour le Thêta-produit, $R \ \bowtie_C \ R' \ \equiv \ \sigma_C(R \times R')$. La figure 3.6 montre un exemple du résultat d'une jointure naturelle entre les deux relations `Vehicule` et `Proprietaire` avec comme condition que l'attribut `proprietaire` de la relation `Vehicule` soit égale à l'attribut `numero` de la relation `Proprietaire`.

R

immatriculation	*marque*	*proprietaire*
3452 AZ 13	BMW	13
9835 EI 07	Renault	76
1234 WW 42	Peugeot	5
9878 DG 94	BMW	76

R $\bowtie_{\text{proprietaire=numero}}$ R'

immatriculation	*marque*	*proprietaire*	*numero*	*nom*	*prenom*	*naissance*
3452 AZ 13	BMW	13	13	Dupont	Marcel	15 mai 1976
9835 EI 07	Renault	76	76	Durand	Pierre	29 novembre 1978
1234 WW 42	Peugeot	5	5	Martin	Paul	4 février 1980
9878 DG 94	BMW	76	76	Durand	Pierre	29 novembre 1978

R'

numero	*nom*	*prenom*	*naissance*
5	Martin	Paul	4 février 1980
13	Dupont	Marcel	15 mai 1976
76	Durand	Pierre	29 novembre 1978

FIGURE 3.6 – Exemple de jointure naturel sur les attributs `proprietaire` et `numero`

Dans cet exemple, la répétition de la même valeur dans les attributs `proprietaire` et `numero` peut être supprimée par une sélection ultérieure.

Deux autres types de jointure existent :

- la **jointure extérieure** qui ajoute au résultat de la jointure naturelle les tuples de chaque relation qui n'ont pas trouvés de correspondance dans l'autre relation, les attributs non remplis du résultat restant alors vides ;
- la **semi-jointure** qui ajoute au résultat de la jointure naturelle les tuples de la première relation qui n'ont pas trouvés de correspondance dans la seconde relation, les attributs non remplis du résultat restant

alors vides. Contrairement aux autres jointures, la semi-jointure n'est pas commutative.

3.2 Traduction d'un modèle EA vers un modèle relationnel

Les modèles Entité-Association et relationnel sont complémentaires car utilisés à des instants différents de la phase de conception d'une base de données. Le modèle Entité-Association permet d'exprimer un Modèle Conceptuel des Données (MCD) pour décrire à un haut niveau d'abstraction les entités à représenter. Le modèle relationnel sert à construire un Modèle Logique de Données (MLD) qui décrit plus précisément et formellement les informations permettant ainsi d'exprimer des requêtes précises grâce à une algèbre, facilitant ainsi le développement dans un langage informatique.

3.2.1 Les règles de transformation

La conception d'un modèle Entité-Association précède celle d'un modèle relationnel qui s'appuie dessus. Le passage de l'un à l'autre est relativement aisé par l'application systématique de 3 règles générales de traduction. Pour simplifier l'exposition des règles de transformation, nous supposons que toutes les associations sont binaires (c'est-à-dire qu'elles ne relient que deux types d'entités entre eux) et que les cardinalités sont de la forme 0:1, 1:1, 0:N ou 0:N. Nous discuterons du cas général ensuite mais ce cas particulier couvre la très grande majorité des diagrammes EA concrets.

Règle 1 : ***Tout type d'entité est traduit par une relation contenant les mêmes attributs et la cl primaire est la clé du type d'entité.***

Par exemple, le type d'entité `Proprietaire` représentée dans la figure 2.6 (p. 16) se traduit par une relation dont le schéma est `Proprietaire`(`nom:string`, `prenom:string`, `naissance:date`). Les deux règles qui suivent s'intéressent aux associations et leur application dépend des cardinalités de part et d'autres de l'association.

Règle 2 : ***Toute association impliquant un type d'entités E avec une cardinalité*** 0:1 ***ou*** 1:1***, et tout autre type d'entité F se traduit par l'ajout, dans la relation résultat de la traduction de E, de la clé de F avec le statut de clé étrangère et l'ajout de tout attribut***

de l'association comme attribut de la relation issue de E. Cette clé étrangère se réfère à la clé primaire de la relation issue de la traduction de F.

Dans le cas où la cardinalité est 0:1, l'attribut ajouté est optionnel, et dans le cas où la cardinalité est 1:1, l'attribut doit être renseigné dans tout tuple de cette relation. Il est possible de préciser cette dernière contrainte dans le langage SQL.

Par exemple, l'association `Possede` qui relie un vehicule à un propriétaire avec la cardinalité 1:1 (cf figure 2.9 p. 18) implique l'ajout dans la relation créée `Vehicule` de la clé de `Proprietaire` comme attributs de manière à obtenir le schéma suivant : `Vehicule`(`immatriculation:string`, `marque:` {Peugeot ; Renault ; BMW}, `nom_proprietaire:string`, `prenom_proprietaire:string`, `naissance_proprietaire:date`).

Règle 3 : ***Toute association entre types d'entités ayant des cardinalités de type $0{:}N$ ou $1{:}N$ de part et d'autre de l'association se traduit par la création d'une relation supplémentaire contenant comme attributs les clés des entités associées ainsi que d'éventuels attributs de l'association. La clé de la relation créée est l'ensemble des attributs représentant les clés des entités associées.***

Par exemple, dans la figure 3.7, l'association reliant les entités `Vehicule` et `Proprietaire` se traduit par une relation dont le schéma est `conduit_par`(`immatriculation:string`, `nom_chauffeur:string`, `prenom_chauffeur:string`, `naissance_chauffeur:date`).

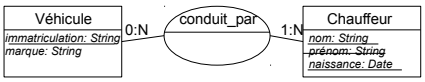


FIGURE 3.7 – Exemple d'entité reliée à elle-même

Cas des autres cardinalités

Dans le cas o la cardinalité est de la forme $x{:}y$ avec x et y deux nombres entiers, plusieurs approches sont possibles. Une approche consiste à appliquer la règle 3 dès lors qu'il n'y a pas de cardinalité 0:1 ou 1:1. Dans

ce cas, le diagramme de la figure 2.10 p. 18) se traduit par deux tables : `Proprietaire`(`nom:string`, `prenom:string`, `naissance:date`) et `est_parent_de`(`nom1:string`, `prenom1:string`, `naissance1:date`, `nom2:string`, `prenom2:string`, `naissance2:date`).

L'autre approche consiste à ajouter à la relation issue de l'entité un nombre d'attributs égal à y multiplié par le nombre d'attributs qui compose la clé de l'entité associée, tous ayant le statut de clé étrangère se référant à la relation issue de la traduction de l'entitè à l'autre extrêmité de l'association. Ainsi, on obtiendrait : `Proprietaire`(`nom:string`, `prenom:string`, `naissance:date`, `nom_pere:string`, `prenom_pere:string`, `naissance_pere:date`, `nom_mere:string`, `prenom_mere:string`, `naissance_mere:date`).

Cas d'un diagramme EA incomplet

Il arrive qu'un diagramme EA soit fourni sans cl, ou que certaine entité ommette leur clé, ou bien que des cardinaliés soient manquantes, ou encore que des types ne soient pas indiqués. Dans le cas où aucune clé n'est mentionné, il est alors courant d'introduire dans la relation correspondante une clé primaire numérique servant uniquement à identifier un unique tuple de cette relation (voir également la section 3.2.2). Tout autre information manquante devrait normalement être exigé de l'auteur du diagramme, mais un jugement au cas par cas permet souvent de deviner la donnée manquante. Un type de données dont on ne peut deviner la nature exact peut être défini comme une chaîne de caractères quelconque. Des cardinalités absentes peuvent être comprises comme une absence de contrainte, que l'on peut remplacer par 0:N.

Cas des relations n-aires

Si la relation n'est pas binaire, on peut appliquer la règle 2 dès lors qu'une seule entité est associé à une cardinalité 0:1 ou 1:1. Il faut alors ajouté à la relation issue de l'entité l'ensemble de toutes les clés des autres entités participant à l'association. La règle 3 se généralise également en l'appliquant dans tout autre cas de figure, par exemple, trois entités ont une cardinalité de type 0:N ou 1:N.

3.2.2 Introduction de clés numériques

La clé d'une entité est parfois composée de plusieurs attributs. Il peut même arriver que la clé contienne l'ensemble des attributs de l'entité. Cette situation introduit de la complexité à plusieurs niveaux : (i) une sélection

simple peut devenir assez longue à écrire pour mentionner tous les attributs de la clé pour retrouver un tuple unique ; (ii) la taille du résultat d'une jointure naturelle est plus importante du faît de la taille de la clé ; (iii) les temps de calcul en sont d'autant plus rallongés.

Pour simplifier l'identifiant d'une entité, on peut introduire artificiellement une clé numérique dont le seul but est d'identifier de manière unique chaque tuple. C'est la solution adoptée dans de nombreux systèmes d'information où, par exemple, à chaque individu est associé un numéro unique (sécurité sociale, numéro d'étudiant, etc).

Dans notre exemple, nous souhaitons remplacer la clé de la relation `Proprietaire` jugée trop longue par une clé numérique. Ceci simplifie les schémas de relation proposés précédemment pour obtenir les schémas suivants :

`Proprietaire`(`numero: int`, `nom: string`, `prenom: string`, `naissance: date`)
`Vehicule`(`immatriculation: string`, `marque:` {Peugeot ; Renault ; BMW}, `no_proprietaire: int`)
`Achete`(`immatriculation: string`, `no_proprietaire: int`, `nom_concessionnaire: string`, `date_achat: string`)

3.3 Dépendances et normalisation

Remarque : cette section mériterait un réaménagement. Certaines explications ne sont pas limpides.

Dans toute base de données non triviale, il existe une multitude de possibilités de modélisation impliquant des choix différents de relations, d'attributs, de clés, de références, etc. La qualité d'une modélisation dépend de plusieurs facteurs tels que la quantité d'information à stocker, la facilité d'expression des requêtes nécessaires à l'utilisation spécifique prévue pour la base de données, la prévention d'erreur de mise à jour et d'incohérences possibles dans la base, etc.

Normaliser une relation consiste à respecter certains critères afin de protéger l'intégrité des données. Selon les critères satisfaits, une **forme normale** est attribuée à la relation, gage de qualité de modélisation. Il existe 5 formes normales (6 avec la forme normale de Boyce-Codd, variante de la troisième forme normale) dont les critères se fondent principalement sur la notion de dépendance fonctionnelle. Il est à noter que les quatrième et cinquième formes

normales sont complexes à mettre en oeuvre et que, dans la pratique, on se contente souvent d'une conception respectant la troisième forme normale. Dans ce document d'introduction aux bases de données, nous ne détaillons pas les quatrième et cinquième formes normales, très peu utilisées.

3.3.1 Les dépendances fonctionnelles

Définition

Une dépendance fonctionnelle (DF) représente le fait qu'à la valeur d'un ou plusieurs attributs, on associe une valeur pour un autre attribut. Une dépendance fonctionnelle signifie que dans la réalité observée un attribut peut être déduit ou calculé à partir d'autres et qu'il est incohérent que plusieurs valeurs existent pour un même ensemble de valeurs de ces autres attributs.

Soit un schéma de relation $R(A_1, A_2, ..., A_n)$ et $X, Y \subseteq \{A_1, A_2, ..., A_n\}$, on note la dépendance fonctionnelle de Y envers X :

$$X \rightarrow Y$$

Cette dépendance fonctionnelle signifie que pour une valeur prise par X, une seule valeur de Y peut exister. Formellement, si $\angle X_i, Y_j\rangle$ et $\langle X_i, Y_k\rangle$ sont les projections sur les attributs X et Y de deux tuples issus de la relation R, la dépendance fonctionnelle $X \rightarrow Y$ implique que $Y_j = Y_k$.

Un exemple est donné dans l'ensemble de tuples représenté par le tableau 3.4.

no_el	nom_eleve	no_prof	nom_prof	matiere	note
1	Dupont	11	Tournesol	Physique	13
1	Dupont	12	Farnsworth	Maths	7
2	Martin	11	Tournesol	Physique	11

TABLE 3.4 – Exemple d'une dépendance fonctionnelle

Ces informations nécessitent deux dépendances fonctionnelles : `no_el` $\rightarrow$ `nom_eleve` et `no_prof` $\rightarrow$ `nom_prof`. Cela signifie que pour un numero d'élève (resp. de professeur) il ne doit correspondre qu'un seul nom d'élève (resp. de professeur). En effet si un tuple associant par exemple un nom de professeur autre que "Tournesol" au numéro "11" était ajouté, cela entrainerait une incohérence dans la base.

Dépendance fonctionnelle élémentaire

Une dépendance fonctionnelle est dite élémentaire si un sous-ensemble de la clé n'est pas source de la dépendance fonctionnelle. Dans une relation de schéma $R(A_1, A_2, ..., A_n)$, une dépendance fonctionnelle $X \rightarrow Y$ avec $X, Y \subseteq \{A_1, A_2, ..., A_n\}$ est dite élémentaire si $\forall a \in X, X - \{a\} \not\rightarrow Y$.

Dépendance fonctionnelle directe

Une dépendance fontionnelle $A \rightarrow B$ est dite directe s'il n'existe pas d'ensemble d'attributs C tel que $A \rightarrow C$ et $C \rightarrow B$.

3.3.2 La première forme normale

Une relation est dite normalisée si un même attribut n'est pas présent plusieurs fois dans une relation et si aucun attribut n'est décomposable en d'autres attributs. Toute relation normalisée est en première forme normale (1FN).

Un exemple de relation en 1FN aurait pour schéma R(`no_el:int`, `no_prof:int`, `matiere:string`, `nom_eleve:string`, `nom_prof:string`, `note:int`). Dans cet exemple, on peut constater que le nom de l'élève et du professeur dépendent des attributs numéro correspondants. Cependant, cette relation en 1FN ne permet d'exprimer que deux dépendances fonctionnelles issues de la clé :

`no_el`, `no_prof` $\rightarrow$ `nom_eleve`
`no_el`, `no_prof` $\rightarrow$ `nom_prof`

L'inconvénient de cette relation en 1FN est qu'elle permet l'ajout de tuple associant différents noms à un même numéro d'élève (ou de professeur). On pourrait sans provoquer d'anomalies créer une relation telle que représentée par la table 3.5. Cette relation contient plusieurs incohérences sur les dépendances entre numéros et noms.

no_el	no_prof	matiere	nom_eleve	nom_prof	note
1	11	Physique	Dupont	Tournesol	17
1	12	Maths	Durand	Farnsworth	12
2	25	Maths	Martin	Farnsworth	4

Table 3.5 – Relation en première forme normale.

3.3.3 La deuxième forme normale

Une relation est en deuxième forme normale (2FN) si :

– elle est en première forme normale ;
– toutes les DF issues de sa clé sont élémentaires.

Un exemple de relation en 2FN aurait pour schéma R(no_el:int, no_prof:int, matiere:string, nom_eleve:string, nom_prof:string, note:int). Cet exemple permet d'exprimer une dépendance fonctionnelle :

no_el → nom_eleve

L'inconvénient de cette relation en 2FN est que des dépendances fonctionnelles issues d'autres attributs que ceux de la clé ne sont pas exprimables. Il est toujours possible d'associer plusieurs noms de professeur à un même numéro sans provoquer d'anomalie. La relation représentée table 3.6 en donne un exemple.

no_el	no_prof	matiere	nom_eleve	nom_prof	note
1	11	Physique	Dupont	Tournesol	17
1	12	Maths	Dupont	Farnsworth	12
2	12	Maths	Martin	Tournesol	4

Table 3.6 – Relation en deuxième forme normale.

3.3.4 La troisième forme normale

Une relation est en troisième forme normale (3FN) si :
– elle est en deuxième forme normale ;
– toutes les DF issues de sa clé sont directes.

Pour respecter la troisième forme normale dans notre exemple, il nous faut concevoir trois schémas de relations : R(no_el:int, nom_eleve:string), R'(no_prof:int, nom_prof:string) et R''(no_el:int, no_prof:int, matiere:string, note:int). Les deux dépendances fonctionnelles allant des numéros d'élève et de professeur vers un nom unique sont satisfaites. La table 3.7 montre un exemple de relations en 3FN.

La troisième forme normale est généralement suffisante pour éviter les anomalies. Il peut néanmoins subsister quelques anomalies si un attribut qui ne fait pas partie de la clé est source d'une DF. Dans notre exemple, cela serait le cas si un professeur ne peut enseigner qu'une matière. La clé de la relation R'' fait qu'un professeur ne peut donner qu'une note à un même élève (et donc dans une seule matière). Par contre il pourrait donner des notes dans différentes matières à différents élèves comme c'est le cas dans l'exemple de la table 3.7 où le professeur Tournesol a donné une note en Physique à l'élève Dupont et une note en Maths à l'élève Martin.

no_el	nom_eleve
1	Dupont
2	Martin

no_prof	nom_prof
11	Tournesol
12	Farnsworth

no_el	no_prof	matiere	note
1	11	Physique	17
1	12	Maths	12
2	11	Maths	4

TABLE 3.7 – Relations en troisième forme normale.

3.3.5 La forme normale de Boyce-Codd

Une relation est en forme normale de Boyce-Codd (F.N.B.C) si :
- elle est en troisième forme normale ;
- la source de chaque DF de la relation est la clé de la relation

La F.N.B.C. est une 3FN spécifique où chaque DF est traduite par une relation. Si plusieurs DF ont la même source, elles peuvent être traduites par une même relation.

Pour être en F.N.B.C, notre exemple consisterait en les relations suivantes : R(`no_el:int`, `nom_eleve:string`), R'(`no_prof:int`, `nom_prof:string`, `matiere:string`), R''(`no_el:int`, `matiere:string`, `note:int`).

3.4 Exercices

Exercice 3 : *Modèle relationnel*

Traduisez le modèle Entité-Association construit lors de l'exercice 2 de la section 2.3 (p. 19) en un modèle relationnel. Identifiez les dépendances fonctionnelles de l'application. Quelle est la forme normale de vos relations ?

Exercice 4 : *Algèbre relationnelle*

Écrivez sous la forme de combinaison d'opérateurs relationnels les requêtes nécessaires pour obtenir sur les relations définies précédemment les informations suivantes :

1. les artistes français ;
2. les artistes américains de genre pop-rock ;
3. les spectacles ayant lieu pendant les prochaines vacances de Nol ;

4. les noms des clients ayant déjà commander des billets ;
5. le genre des artistes italiens ;
6. le prix des billets des concerts de Metallica ;
7. les artistes que Pierre Dupond a vu au Transbordeur ;
8. les spectacles pour lesquels aucune place n'a été encore vendue ;
9. pour dtecter des conflits entre spectacles, nous voudrions connaitre la liste des spectacles ayant lieu simultanment au mme endroit ;
10. quelle salles de spectacle vend des billets plus chers que tous les billets vendus au Zénith ?

Chapitre 4

Le langage SQL

Le Langage SQL (*Structured Query Language*) est un langage informatique de gestion de BDD relationnelles. SQL a fait l'objet de plusieurs normalisations, la première datant de 1986. SQL regroupe trois catégories de langage :

- c'est un Langage de Description de Données (LDD) permettant de décrire les objets de la base sous forme de table (l'équivalent des relations dans le modèle relationnel).
- c'est un Langage de Manipulation de Données (LMD) permettant d'interroger et de modifier le contenu d'une BDD.
- c'est un Langage de Contrôle d'accès aux Données (LCD) permettant de gérer les droits d'accès aux données des utilisateurs.

La syntaxe de SQL est relativement simple. Ce chapitre décrit les instructions nécessaires à la définition d'une BDD, à la modification de son contenu puis à son interrogation.

4.1 Définition de données

Les relations du modèle relationnel sont représentées en SQL sous la forme de table.

Remarque : les annexes de ce livret contiennent une copie de diapositives comprenant des exemples supplémentaires et plus graphiques des requêtes SQL.

4.1.1 Création de tables

L'instruction de création de table en SQL est `CREATE TABLE`. Sa syntaxe est décrite ci-dessous, en utilisant des crochets pour indiquer qu'une partie est optionnelle.

`CREATE TABLE` *nom_table*
(*nom_col1* TYPE1 [[nom_contrainte_1] contrainte_1],
nom_col2 TYPE2 [[nom_contrainte_2] contrainte_2],
...,
[autre_contrainte_1],
[autre_contrainte_2],
...) ;

Cette instruction créé une table appelée *nom_table* composée de plusieurs colonnes dont les noms sont *nom_col1*, *nom_col2*, etc. Chacune de ces colonnes a un type (TYPE1, TYPE2, ...). La table 4.1 présente quelques types fréquemment utilisés.

Type de données	**Description**
`INT`[(*longueur*)]	entier
`FLOAT`[(*longueur,précision*)]	réel
`CHAR`(*longueur*)	chaîne de caractères de longueur fixe pouvant aller jusqu'à 255 caractères
`VARCHAR`(*longueur*)	chaîne de caractères de longueur variable pouvant aller jusqu'à 2000 caractères
`DATE`	format de date

TABLE 4.1 – Quelques types de données des tables SQL

À chaque colonne peut être attribué des contraintes d'intégrité. Ces contraintes sont optionnelles. Pour affecter une contrainte à une colonne on utilise un éventuel nom de contrainte (facultatif) et un type de contrainte. La table 4.2 présente quelques contraintes fréquemment utilisées.

Par exemple la création de la table `Proprietaire` telle que décrite par la relation située section 3.2.2 peut se faire par l'instruction :

```
CREATE TABLE Proprietaire
       (numero INT PRIMARY KEY,
        nom VARCHAR(50) NOT NULL,
        prenom VARCHAR(30),
        naissance DATE);
```

Contrainte d'intégrité	description
PRIMARY KEY	la colonne est la clé primaire (doit être unique à chaque enregistrement).
NOT NULL	l'attribut ne peut pas être sans valeur.
DEFAULT *valeur*	*valeur* est la valeur par défaut de l'attribut si aucune ne lui est donnée.

TABLE 4.2 – Quelques contraintes d'intégrité des tables SQL

À sa création la table `Proprietaire` sera vide.

Il est également possible d'ajouter des contraintes supplémentaires à la table elle-même. L'une des contraintes très classiques est la définition de clés étrangères comme suit :

```
CREATE TABLE Vehicule
       (immatriculation CHAR(8) PRIMARY KEY,
        modele VARCHAR(50) NOT NULL,
        proprietaire INT,
        CONSTRAINT fk_proprietaire FOREIGN KEY(proprietaire)
        REFERENCES Proprietaire(numero));
```

Ceci indique qu'une contrainte appelée `fk_proprietaire` de type `FOREIGN KEY` existe pour l'attribut `proprietaire`

4.1.2 Suppression et modification de tables

Une table peut être renommée par l'instruction `RENAME` selon la syntaxe suivante :

`RENAME TABLE` *ancien_nom* `TO` *nouveau_nom* ;

Une table peut être supprimmée par l'instruction `DROP TABLE` selon la syntaxe suivante :

`DROP TABLE` *nom_table* ;

La table et son contenu seront détruits par cette opération.

Enfin, la structure d'une table peut être modifiée par l'instruction `ALTER TABLE` selon la syntaxe suivante :

```
ALTER TABLE nom_table
    ADD (nom_col1 col1_definition),
    ...,
    MODIFY nom_col2 col2_definition,
    ...,
    DROP nom_col3,
    ...,
    ... ) ;
```

Cette instruction peut ajouter des colonnes à la table (ici *nom_col1*) définies comme lors d'un `CREATE TABLE` dans *col1_definition*. Les enregistrements existants auront des valeurs nulles sur les colonnes ajoutées. Elle peut modifier une colonne (ici *nom_col2*) pour opter pour une nouvelle définition (spécifiée dans *col2_definition*). Elle peut supprimer des colonnes (ici *nom_col3*). Il est également possible de modifier ou d'ajouter des contraintes en écrivant par exemple `ADD CONSTRAINT` puis suivre la même syntaxe qu'à la création de table.

4.2 Modification de données

Une fois les tables créées, leur contenu peut être modifié en ajoutant, supprimant ou modifiant des données sous la forme d'enregistrements (les tuples en modèle relationnel).

4.2.1 Insertion de données

L'instruction `INSERT` sert à ajouter des enregistrements à une table avec la syntaxe suivante :

```
INSERT INTO nom_table[(nom_col1,nom_col2,...)]
    VALUES (val1,val2,...), (val3,val4,...) ;
```

Cette instruction ajoute plusieurs enregistrements avec comme valeur d'attributs celles qui sont précisées après `VALUES`. Si des noms de colonnes sont précisées après le nom de la table, les valeurs doivent être écrites dans le même ordre que les noms de colonne. Si les noms de colonne ne sont pas précisées, ces valeurs sont affectées aux colonnes selon leur ordre lors de la création de la table.

Par exemple, pour ajouter deux enregistrements à la table `Proprietaire` on peut écrire :

```
INSERT INTO Proprietaire
    VALUES (1,"Martin","Paul","1982-11-05"),
           (2,"Durand","Marcel","1976-11-23");
```

4.2.2 Suppression de données

L'instruction `DELETE` sert à supprimer des enregistrements d'une table avec la syntaxe suivante :

`DELETE FROM` *nom_table*
  `WHERE` *condition* ;

La *condition* sert à sélectionner les enregistrements à supprimer (comme lors de l'utilisation de l'opérateur relationnel de sélection). Par exemple pour supprimer tous les enregistrements de la table *Proprietaire* dont le nom est "Martin" on écrirait :

```
DELETE FROM Proprietaire
     WHERE nom="Martin";
```

4.2.3 Mise à jour de données

L'instruction `UPDATE` sert à modifier la valeur d'attributs d'enregistrements existants dans une table avec la syntaxe suivante :

`UPDATE` *nom_table*
  `SET` *nom_col1* = *expression1*,
    nom_col2 = *expression2*,
    ...
  `WHERE` *condition* ;

Cette instruction modifie les attributs *nom_col1*, *nom_col2*, etc pour leur donner comme valeurs *expression1*, *expression2*, ... dans les enregistrements qui satisfont la condition *condition*.

Par exemple, pour affecter la date de naissance "1978-12-15" à tous les enregistrements dont le nom est "Martin", on peut écrire :

```
UPDATE Proprietaire
      SET naissance = "1978-12-15"
      WHERE nom = "Martin";
```

4.3 Interrogation de données

Toutes les requêtes d'interrogation de la BDD pour retrouver des informations se font par l'instruction `SELECT`. La syntaxe de la commande `SELECT`, présentée ci-dessous, permet beaucoup de possibilités.

`SELECT` [`DISTINCT`] *attr1* [*nom_attr1*],*attr2* [*nom_attr2*],...
  `FROM` *nom_table1*,*nom_table2*,...
  [`WHERE` *condition*]
  [`GROUP BY` *expr1*, *xpr2*, ...] [`HAVING` *condition*]
  [`ORDER BY` *attr1*, *attr2*, ... [`DESC`]]

L'ensemble des noms de table nécessaires à la requête figurent après l'instruction `FROM`. Les tables nécessaires ne sont pas forcément que celles qui contiennent le résultat à trouver. Elles peuvent être utilisées pour exprimer des jointures.

L'ensemble *attr1*,*attr2*,... contient les attributs dont on veut connaître la valeur. Dans le résultat ces attributs peuvent être renommés si un nom est précisé juste après (ici *nom_attr1*, *nom_attr2*, ...). On peut préciser la table d'où vient chaque attribut en écrivant *nom_table.attribut*, notamment s'il peut y avoir des ambiguïtés entre des tables ayant des attributs de même nom. L'instruction optionnelle `DISTINCT` supprime les doublons dans le résultat. Le symbole * représente l'ensemble des attributs d'une table ou de toutes les tables de la requête.

Les conditions après l'instruction `WHERE` sélectionne uniquement les enregistrements des tables qui satisfont la *condition*. Cette condition peut être complexe et composée de plusieurs critères reliés par l'instruction `AND` (pour exprimer une conjonction) ou `OR` (pour exprimer une disjonction).

L'instruction `ORDER BY` ordonne le résultat selon la valeur de certains attributs. C'est-à-dire l'ordre dans lequel seront présentés les enregistrements résultats de la requête. Plusieurs attributs peuvent y figurer pour trier d'abord en fonction d'un premier attribut, puis d'un second, et ainsi de suite. Par défaut, le tri est ascendant. Pour un tri descendant il faut ajouter l'instruction `DESC`.

Les instructions `GROUP BY` et `HAVING` sont expliquées en section 4.3.4.

4.3.1 Expression de projection

L'opérateur relationnel de projection est réalisé par la précision après l'instruction `SELECT` des attributs sur lesquels s'effectue la projection. Par exemple l'instruction :

```
SELECT *
     FROM Proprietaire;
```

n'effectue aucune projection car elle conserve l'ensemble des attributs de la table `Proprietaire`. Pour effectuer une projection sur les attributs nom et prenom on écrit :

```
SELECT nom, prenom
     FROM Proprietaire;
```

4.3.2 Expression de sélection

L'opérateur relationnel de sélection est réalisé par la précision après l'instruction `WHERE` de la condition utilisée pour sélectionner des enregistrements. Par exemple pour sélectionner les enregistrements où l'attribut naissance a pour valeur une date ultérieure au 1er janvier 1980 on écrit :

```
SELECT *
     FROM Proprietaire
     WHERE naissance > "1980-01-01";
```

Plusieurs conditions peuvent être combinées, séparées par le mot-clé `AND`. Pour exprimer des conditions, on peut utiliser les opérateurs suivants :

- les opérateurs mathématiques de base =, ! =, <, >, <=, >= pour comparer un attribut à un autre attribut ou à une valeur représentées par un nombre ou une chaîne de caractères ;
- l'instruction *expr1* `BETWEEN` *expr2* `AND` *expr3* qui teste si un attribut est entre deux valeurs ;
- l'instruction *expr1* `IN` (*expr2*, *expr3*, ...) qui teste si un attribut est parmi un ensemble de valeurs ;
- l'instruction *expr1* `LIKE` *chaine* qui teste le format d'une chaîne de caractères, défini par *chaine* où le symbole _ peut remplacer n'importe quel caractère et % un nombre quelconque de caractères.

Par exemple pour retrouver tous les propriétaires dont le nom commence par M on écrit :

```
SELECT *
     FROM Proprietaire
     WHERE nom LIKE 'M%';
```

4.3.3 Expression de jointure

L'expression de jointures se fait simplement par des conditions d'égalité d'attributs entre plusieurs tables de la requête. Par exemple, si l'on souhaite connaître les noms, prénoms et numéro d'immatriculation de tous les propriétaires de véhicules, il faut réaliser une jointure entre les relations `Proprietaire` et `Vehicule` suivie par une projection. En SQL, on l'exprime de la manière suivante :

```
SELECT prenom, nom, immatriculation
     FROM Proprietaire, Vehicule
     WHERE Proprietaire.numero = Vehicule.no_proprietaire;
```

4.3.4 Interrogations de groupe

Fonctions de groupe

Des fonctions de groupe permettent de faire des calculs sur l'ensemble des valeurs d'une colonne. Le tableau 4.3 présente les principales.

Fonction	Description
`SUM`(*attribut*)	renvoie la somme des valeurs numériques d'un attribut pour tous les enregistrements sélectionnés
`COUNT`(*)	renvoie le nombre d'enregistrements sélectionnés
`MAX`(*attribut*)	renvoie la valeur maximale d'un attribut pour tous les enregistrements sélectionnés
`MIN`(*attribut*)	renvoie la valeur minimale d'un attribut pour tous les enregistrements sélectionnés
`AVG`(*attribut*)	renvoie la valeur moyenne d'un attribut pour tous les enregistrements sélectionnés

TABLE 4.3 – Fonctions de groupe en SQL

Par exemple pour connaître le nombre de voitures que possède le propriétaire dénommé "Paul Martin" on écrit :

```
SELECT COUNT(*)
     FROM Proprietaire, Vehicule
     WHERE Proprietaire.numero = Vehicule.no_proprietaire
     AND Proprietaire.nom = "Martin"
     AND Proprietaire.prenom = "Paul";
```

Instruction `GROUP BY`

Dans l'exemple précédent et en l'absence d'indications contraires, l'ensemble des enregistrements traités est considéré comme appartenant à un unique groupe. Ici c'est le groupe des voitures dont le propriétaire est Paul Martin qui est considéré. La requête renvoie le nombre de membres de ce groupe.

Il est possible d'appliquer dans une même requête une fonction de groupe à plusieurs groupes. Pour cela, il est nécessaire d'indiquer par l'instruction `GROUP BY` *expr1*,*expr2*,... le critère de regroupement des réponses. Ce critère désigne les attributs de la réponse utilisés pour rassembler dans un même groupe tous les enregistrements ayant la même valeur pour cet attribut. Par exemple, pour connaître le nombre de véhicules possédés par chaque propriétaire, représentés par leur nom et prénom, on écrit :

```
SELECT Proprietaire.nom, Proprietaire.prenom, COUNT(*)
     FROM Proprietaire, Vehicule
     WHERE Proprietaire.numero = Vehicule.no_proprietaire
     GROUP BY Proprietaire.nom, Proprietaire.prenom;
```

Pour comprendre comment fonctionne la fonction de groupe, il faut d'abord examiner les clauses `FROM` et `WHERE` : celles-ci indiquent que l'on s'intéresse uniquement aux combinaisons de propriétaires et de véhicules pour lesquelles le `numero` (du propriétaire) est gale au `no_proprietaire` (du véhicule). Ceci donne l'ensemble de résultat que l'on va traiter. Ensuite, la clause `GROUP BY` indique que l'on partitionne l'ensemble des résultats en groupant ensemble les lignes ayant le même `nom` et `prenom`. Enfin, pour chaque groupe dans la partition (pouvant contenir plusieurs lignes puisqu'un propriétaire peut avoir plusieurs voitures) on effectue l'opération indiquée par la fonction de groupe dans la clause `SELECT`. Ici, la fonction est `COUNT` qui donne simplement le nombre de lignes dans le groupe. Enfin, puisque chaque groupe contient un et un seul couple (`nom`,`prenom`), on peut également demander ce qu'ils sont afin de savoir à qui est associé le nombre calculé par `COUNT`.

Notons que, dans cette requête, tous les homonymes seront considérés dans un même groupe. Par exemple, la requête renvoie le nombre de véhicules appartenant à tous les Paul Martin de la base (pour éviter cela il aurait plutôt fallu opérer un regroupement selon la clé de la table Proprietaire, le *numero*). Il est donc possible d'afficher ce nombre total de véhicules, ainsi que les attributs *nom* et *prenom* car tous les enregistrements d'un même groupe ont des valeurs identiques pour ces attributs. Par contre, il ne serait pas possible d'afficher l'attribut *naissance* du groupe car des dates de naissance différentes peuvent exister dans un groupe.

Instruction `HAVING`

Lors d'un regroupement, l'instruction `WHERE` permet de sélectionner des enregistrements **avant** d'opérer le regroupement. Pour effectuer une sélection sur des groupes, on peut utiliser l'instruction `HAVING` *condition*, où *condition* décrit la condition à respecter pour qu'un groupe soit sélectionné. Si l'on souhaite modifier la requête précédente pour ne garder dans le résultat que les propriétaires ayant au moins 2 véhicules on écrira :

```
SELECT Proprietaire.nom, Proprietaire.prenom, COUNT(*)
     FROM Proprietaire, Vehicule
     WHERE Proprietaire.numero = Vehicule.no_proprietaire
     GROUP BY Proprietaire.nom, Proprietaire.prenom
     HAVING COUNT(*) > 1;
```

4.4 Imbrication de requêtes

L'instruction `SELECT` peut être utilisée de manière imbriquée à d'autres requêtes pour exprimer des conditions, réaliser des insertions ou même lors de la création d'une table.

4.4.1 Définition de données par requêtes imbriquées

Une insertion peut s'appuyer sur une requête `SELECT` pour récupérer dans d'autres tables des valeurs d'attributs à insérer dans une table. Par exemple, pour insérer dans la table `Proprietaire` un ensemble d'enregistrement issus d'une table `Employe` on écrit :

```
INSERT INTO Proprietaire
    SELECT numero, nom, prenom, naissance
                    FROM Employe;
```

Ces données peuvent être également insérées automatiquement à la création de la table si on l'écrit de la manière suivante :

```
CREATE TABLE Proprietaire
       (numero INT PRIMARY KEY,
        nom VARCHAR(50) NOT NULL,
        prenom VARCHAR(30),
        naissance DATE)
AS SELECT numero, nom, prenom, naissance
        FROM Employe;
```

4.4.2 Sélections imbriquées

Un critère de recherche exprimé dans une clause `WHERE` peut s'appuyer sur le résultat d'une autre requête.

Requête imbriquée à résultat unique

Le cas le plus simple est l'utilisation d'une requête renvoyant un unique résultat comme dans l'exemple ci-dessous :

```
SELECT prenom, nom
     FROM Proprietaire
     WHERE naissance = (SELECT MIN(naissance)
                        FROM Proprietaire);
```

La requête imbriquée renvoie la plus petite date de naissance des propriétaires enregistrés dans la base. La requête globale l'utilise pour trouver les prénoms et noms du (ou des) propriétaires correspondants.

L'utilisation d'une requête imbriquée renvoyant une seule ligne mais composée de plusieurs attributs est possible. La clause WHERE fait alors apparaître entre parenthèses l'ensemble des attributs à comparer avec le résultat de la requête imbriquée. Un exemple est donné ci-dessous.

```
SELECT immatriculation
     FROM Vehicule
     WHERE (marque,no_proprietaire) = (SELECT marque,no_proprietaire
                        FROM Proprietaire,Vehicule
                        WHERE numero = no_proprietaire
                        AND immatriculation = '1234 XX 42');
```

Dans cet exemple, la requête imbriqué renvoie la marque et le numero de propriétaire du véhicule immatriculé “1234 XX 42”. Le résultat de cette requête est alors utilisé pour retrouver les immatriculations de tous les véhicules appartenant au même propriétaire et ayant la même marque.

Requête imbriquée à résultat multiple

Dans le cas où la requête imbriquée renvoie plusieurs lignes, on peut utiliser des opérateurs de comparaison qui compare un attribut à un ensemble de valeurs. Par exemple, pour trouver le propriétaire dont la date de naissance est plus petite que celle de tous les autres propriétaires on utilise l'instruction `ALL` de la manière suivante :

```
SELECT prenom, nom
     FROM Proprietaire
     WHERE naissance <= ALL (SELECT naissance
                             FROM Proprietaire);
```

L'instruction `ANY` s'utilise de la même manière et représente une condition satisfaite si au moins une des lignes renvoyées vérifie la comparaison.

L'instruction `IN` teste si l'attribut appartient à l'ensemble renvoyé par la requête imbriquée. Pour retrouver les noms de tous les propriétaires de véhicules de la marque Peugeot on peut ainsi écrire :

```
SELECT nom
     FROM Proprietaire
     WHERE numero IN (SELECT no_proprietaire
                      FROM Vehicule
                      WHERE marque = 'Peugeot');
```

Enfin, l'instruction `EXISTS` teste l'existence d'au moins un résultat dans la requête imbriquée. Pour obtenir l'ensemble des véhicules appartenant à des propriétaires nés après 1980, on peut écrire :

```
SELECT immatriculation
     FROM Vehicule v
     WHERE EXISTS (SELECT *
                   FROM Proprietaire
                   WHERE naissance >= "1980-01-01"
                   AND numero = v.no_proprietaire);
```

Notons dans cet exemple la référence qui est faite depuis la requête imbriquée à une table de la requête principale. Ceci est possible en nommant la table Vehicule dans la requête principale par le symbole v puis en utilisant ce symbole pour accéder à l'attribut `no_proprietaire` de cette table.

Références

Pour aller plus loin, voici quelques références qui ont servi d'inspiration pour la rédaction de ce livret.

Livres et chapitre de livre :

- Bases de donnés relationnelles et Web, In : *Introduction à la science informatique*, Gilles Dowek, 2011. Disponible en ligne à `https://wiki.inria.fr/wikis/sciencinfolycee/images/2/2e/LSICh7.pdf`
- *Les bases de données relationnelles*, A. Flory, F. Laforest, ed. Economica, 2005
- *Des bases de données à l'Internet*, P. Mathieu, ed. Vuibert, 2000
- *Bases de données et systèmes dinformation*, N. Boudjlida, ed. Dunod, 1999
- *Foundations of Databases*, S. Abiteboul, 1995. La bible des chercheurs en bases de données. `http://webdam.inria.fr/Alice/`

Sites Internet (accessibles au moment de la rédaction de ce livret) :

- `http://www.metz.supelec.fr/metz/personnel/vialle/course/SI/index.htm`
- `http://www.crescenzo.nom.fr/CMBasesDeDonnees/`
- `http://www.emse.fr/~vercouter/cours/csi/`
- `http://www.volle.com/rennes2/index.htm`

Index

algèbre relationnel, 21
ALL, 51
ALTER TABLE, 43
analyse, 12
ANY, 52
association, 16
attribut, 15

Base de Données, 7

cardinalité, 18
clé, 15
complément, 29
conception, 12
contrainte d'intégrité, 42
CREATE TABLE, 42, 50
cycle de développement, 11

dépendance fonctionnelle, 35
dépendance fonctionnelle élémentaire, 36
dépendance fonctionnelle directe, 36
DELETE, 45
deuxième forme normale, 36
différence, 26
division, 29
DROP TABLE, 43

enregistrement, 44
entité, 14
EXISTS, 52

forme normale, 34
forme normale de Boyce-Codd, 38

GROUP BY, 49

HAVING, 50

identifiant, 14
implémentation, 12
IN, 52
INSERT, 44
intersection, 29

jointure, 30, 48
jointure extérieure, 30
jointure naturelle, 30

mise en œuvre, 12
modèle conceptuel de données, 13
modèle logique de données, 21

n-uplet, 22

occurrence, 15
opérateurs relationnels, 24

première forme normale, 36
produit cartésien, 27
projection, 25, 46
propriété, 14

relation, 14, 21
relation normalisée, 36
RENAME, 43
restriction, 25

sélection, 25, 47
SELECT, 46, 50
semi-jointure, 30
SGBD, 8
SQL, 41

Système d'Information Informatisé, 7
Système d'Information Opérationnel, 7
Système de gestion de bases de données, 8

table, 41
Thêta-produit, 29
troisième forme normale, 37
tuple, 22

union, 26
UPDATE, 45

Querying Databases

- What is a query?
  - an **expression** in relational algebra, i.e., **operations** on tables
  - The result of a query on one or more table(s) is a **table**
- Operations for filtering:
  - **Selection** (filter / criteria) = selection of rows
  - **Projection** = selection of columns
- Operations for combining 2 tables:
  - **Cartesian product** (all possibilities)
  - **Join** (bridging between related rows)
  - **Union** (union of sets of rows)
  - **Difference** (elimination of rows)

Relational Algebra Operations

- **Unary** Operations $\text{UnOp}\ (\ \text{Table}_1\ ) \rightarrow \text{Table}_2$
  - Selection (or restriction) : choice of rows
  - Projection : choice of columns
- **Binary** Operations $(\ \text{Table}_1\ )\ \text{BinOp}\ (\ \text{Table}_2\ ) \rightarrow \text{Table}_3$
  - On tables with the same structure: set-operators
    - Union
    - Intersection
    - Difference
  - On tables with different structure
    - Cartesian Product
    - Join

Some definitions

- A **table** is a set of rows
- A **row** L is a sequence of values V_i, $L = (V_1, \dots, V_i, \dots, V_n)$
- The **row** (V_i) built from L is denoted $L(i)$
- Let be two rows $L_1 = (V_1, \dots, V_n)$ and $L_2 = (W_1, \dots, W_m)$, the **concatenation** of L_1 and L_2, denoted L_1L_2 is:

$$L_1L_2 = (V_1, \dots, V_n, W_1, \dots, W_m)$$

Selection

Choice of rows according to a Boolean condition

$$\sigma_{\text{condition}}(\text{ Table })$$

Selection

Choice of rows according to a Boolean condition

$$\sigma_{\text{condition}}(\text{ Table })$$

Person

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Clark K.	35	Metropolis, USA	Krypton
Lois L.	28	Metropolis, USA	Metropolis, USA
Tarzan	25	London, UK	Forest, Benin

- People living in London:

$$\sigma_{\text{Address="London, UK"}}(\text{Person})$$

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Tarzan	25	London, UK	Forest, Benin

Selection

Choice of rows according to a Boolean condition

$$\sigma_{\text{condition}}(\text{ Table })$$

Person

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Clark K.	35	Metropolis, USA	Krypton
Lois L.	28	Metropolis, USA	Metropolis, USA
Tarzan	25	London, UK	Forest, Benin

- People living in London:

$$\sigma_{\text{Address="London, UK"}}(\text{Person})$$

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Tarzan	25	London, UK	Forest, Benin

- People living where they were born:

$$\sigma_{\text{Address=Birthplace}}(\text{Person})$$

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Lois L.	28	Metropolis	Metropolis

Selection (contd.)

- Persons being less than 30 y.o.:

$\sigma_{Age<30}(Person)$

Name	Age	Address	Birthplace
Lois L.	28	Metropolis	Metropolis
Tarzan	25	London, UK	Forest, Benin

this is a table

Selection (contd.)

- Persons being less than 30 y.o.:

$\sigma_{Age<30}(Person)$

Name	Age	Address	Birthplace
Lois L.	28	Metropolis	Metropolis
Tarzan	25	London, UK	Forest, Benin

this is a table too!

Selection (contd.)

- People less than 30 y.o.:

Name	Age	Address	Birthplace
Lois L.	28	Metropolis	Metropolis
Tarzan	25	London, UK	Forest, Benin

$$\sigma_{Age<30}(Person)$$

- **Composition of selection**: people less than 30 living in London:

$$\sigma_{Address="London,\ UK"}(\sigma_{Age<30}(Person))$$

Name	Age	Address	Birthplace
Tarzan	25	London, UK	Forest, Benin

Selection (contd.)

- People less than 30 y.o.:

Name	Age	Address	Birthplace
Lois L.	28	Metropolis	Metropolis
Tarzan	25	London, UK	Forest, Benin

$$\sigma_{Age<30}(Person)$$

- **Composition of selection**: people less than 30 living in London:

$$\sigma_{Address="London,\ UK"}(\sigma_{Age<30}(Person))$$

- **Conjunction of selection**:

$$\sigma_{Address="London,\ UK"\ and\ Age<30}(Person))$$

equivalent to the composition

Name	Age	Address	Birthplace
Tarzan	25	London, UK	Forest, Benin

Formalism for the selection operation

- Selection / constant:

$\sigma_{i = a}(T)$ is the set of rows L of the table T where $L(i) = a$

$$\sigma_{i = a}(T) = \{ L \mid L \in T \text{ and } L(i) = a \}$$

- Selection / inter-columns:

$\sigma_{i = j}(T)$ is the set of rows L of the table T where $L(i) = L(j)$

$$\sigma_{i = j}(T) = \{ L \mid L \in T \text{ and } L(i) = L(j) \}$$

- Comparison operators $\neq$, $<$, $>$, $\leq$, $\geq$ can be used
- Boolean operators AND, OR

Projection

Choice of columns in a table

$$\pi_{\text{column(s)}} (\text{Table})$$

Projection

Choice of columns in a table

$$\pi_{\text{column(s)}}(\text{ Table })$$

Person

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Clark K.	35	Metropolis, USA	Krypton
Lois L.	28	Metropolis, USA	Metropolis, USA
Tarzan	25	London, UK	Forest, Benin

- Name and age of people:

$$\pi_{\text{Name,Age}}(\text{Person})$$

Name	Age
James B.	47
Clark K.	35
Lois L.	28
Tarzan	25

Projection (contd.)

- Name of people living in London:

$$\pi_{\text{Name}}(\sigma_{\text{Address="London, UK"}}(\text{Person}))$$

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Tarzan	25	London, UK	Forest, Benin

Projection (contd.)

- Name of people living in London:

$$\pi_{\text{Name}}(\sigma_{\text{Address="London, UK"}}(\text{Person}))$$

Name	Age	Address	Birthplace
James B.	47	London, UK	London, UK
Tarzan	25	London, UK	Forest, Benin

Name
James B.
Tarzan

Formalism for the projection operation

- Projection onto columns:

$\pi_{i_1,\ldots,i_k}(T)$ is the set of rows of the table T obtained by keeping columns $i_1, i_2, \ldots, i_k$

$$\pi_{i_1,\ldots,i_k}(T) = \{ \, (L(i1), \ldots, L(ik)) \mid L \in T \, \}$$

The result of any operation on tables is also a table
⇒ different operations can be combined

The result of selections or projection
can be **an empty table**

Cartesian Product

Each row of the first table is combined with all the rows of the second table

$$\text{Table}_1 \otimes \text{Table}_2$$

Cartesian Product

Each row of the first table is combined with all the rows of the second table

$$\text{Table}_1 \otimes \text{Table}_2$$

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

Woman

Name	Address
Lois L.	Metropolis
Catwoman	Gotham City

- All man-woman pairs:

$$\text{Man} \otimes \text{Woman}$$

Name_1	Address_1	Name_2	Address_2
James B.	London	Lois L.	Metropolis
James B.	London	Catwoman	Gotham City
Clark K.	Metropolis	Lois L.	Metropolis
Clark K.	Metropolis	Catwoman	Gotham City
Tarzan	London	Lois L.	Metropolis
Tarzan	London	Catwoman	Gotham City

Cartesian Product (contd.)

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

City

CName	Population	Country
London	7,556,900	UK
Metropolis	8,391,881	USA

- In which country do the men of the Man table live?

1) $\text{Man} \otimes \text{City}$

Name	Address	CName	Population	Country
James B.	London	London	7,556,900	UK
James B.	London	Metropolis	8,391,881	USA
Clark K.	Metropolis	London	7,556,900	UK
Clark K.	Metropolis	Metropolis	8,391,881	USA
Tarzan	London	London.	7,556,900	UK
Tarzan	London	Metropolis	8,391,881	USA

Cartesian Product (contd.)

2) Followed by a selection:

$$\sigma_{\text{Address=CName}}(\text{Man} \otimes \text{City})$$

Name	Address	CName	Population	Country
James B.	London	London	7,556,900	UK
James B.	London	Metropolis	8,391,881	USA
Clark K.	Metropolis	London	7,556,900	UK
Clark K.	Metropolis	Metropolis	8,391,881	USA
Tarzan	London	London	7,556,900	UK
Tarzan	London	Metropolis	8,391,881	USA

Name	Address	CName	Population	Country
James B.	London	London	7,556,900	UK
Clark K.	Metropolis	Metropolis	8,391,881	USA
Tarzan	London	London	7,556,900	UK

Cartesian Product (contd.)

- Man-woman pairs living in the same city:

$$\sigma_{Address_1=Adrress_2}(Man \otimes Woman)$$

$Name_1$	$Address_1$	$Name_2$	$Address_2$
James B.	London	Lois L.	Metropolis
James B.	London	Catwoman	Gotham City
Clark K.	Metropolis	Lois L.	Metropolis
Clark K.	Metropolis	Catwoman	Gotham City
Tarzan	London	Lois L.	Metropolis
Tarzan	London	Catwoman	Gotham City

Cartesian Product (contd.)

- Man-woman pairs living in the same city:

$$\sigma_{Address_1=Adrress_2}(Man \otimes Woman)$$

$Name_1$	$Address_1$	$Name_2$	$Address_2$
James B.	London	Lois L.	Metropolis
James B.	London	Catwoman	Gotham City
Clark K.	Metropolis	Lois L.	Metropolis
Clark K.	Metropolis	Catwoman	Gotham City
Tarzan	London	Lois L.	Metropolis
Tarzan	London	Catwoman	Gotham City

$Name_1$	$Address_1$	$Name_2$	$Address_2$
Clark K.	Metropolis	Lois L.	Metropolis

Formalism for the cartesian product operation

- $T_1 \otimes T_2$ is the set of all rows that can be obtained by concatenating a row of table T_1 with a row of table T_2

$$T_1 \otimes T_2 = \{ L_1 L_2 \mid L_1 \in T_1 \text{ and } L_2 \in T_2 \}$$

The cartesian product $T_1 \otimes T_2$ is often followed by a selection in which one column from T_1 and one column from T_2 are equalled

- So much so that there is an operation for this called **Join**

Join

Cartesian product followed by a selection

$$\text{Table}_1 \underset{\text{condition}}{\bowtie} \text{Table}_2$$

Join

Cartesian product followed by a selection

$$\text{Table}_1 \underset{\text{condition}}{\bowtie} \text{Table}_2$$

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

City

CName	Population	Country
London	7,556,900	UK
Metropolis	8,391,881	USA

- In which country do the men of the Man table live?

$$\text{Man} \underset{\text{Address=CName}}{\bowtie} \text{City} \Leftrightarrow \sigma_{\text{Address=CName}}(\text{Man} \otimes \text{City})$$

Name	Address	CName	Population	Country
James B.	London	London	7,556,900	UK
Clark K.	Metropolis	Metropolis	8,391,881	USA
Tarzan	London	London.	7,556,900	UK

Join (contd.)

- Couple man-woman of the same city

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

Woman

Name	Address
Lois L.	Metropolis
Catwoman	Gotham City

$$\text{Man} \underset{\text{Address}_1\text{=Address}_2}{\bowtie} \text{Woman} \Leftrightarrow \sigma_{\text{Address}_1\text{=Address}_2}(\text{Man} \otimes \text{Woman})$$

Name_1	Address_1	Name_2	Address_2
Clark K.	Metropolis	Lois L.	Metropolis

Formalism for the join operation

- $T_1 \underset{i=j}{\bowtie} T_2$ is the set of rows obtained by concatenating a row of table T_1 with a row of table T_2 such that $L_1(i) = L_2(j)$

$$T_1 \underset{i=j}{\bowtie} T_2 = \{ L_1L_2 \mid L_1 \in T_1 \text{ and } L_2 \in T_2 \text{ and } L_1(i) = L_2(j) \}$$

- Link with the Cartesian product and the selection:
  - Let m be the number of columns of $T1$

$$T_1 \underset{i=j}{\bowtie} T_2 = \sigma_{i=j+m}(T_1 \otimes T_2)$$

As for selection, comparison operators $\neq$, $<$, $>$, $\leq$, $\geq$ can be used as well as boolean operators

Union

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

Woman

Name	Address
Lois L.	Metropolis
Catwoman	Gotham City

- The set of people:

Man ∪ Woman

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London
Lois L.	Metropolis
Catwoman	Gotham City

Union

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

Woman

Name	Address
Lois L.	Metropolis
Catwoman	Gotham City

- The set of people:

  Man ∪ Woman

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London
Lois L.	Metropolis
Catwoman	Gotham City

- People living in Metropolis:

$$\sigma_{\text{Address="Metropolis"}}(\text{Man} \cup \text{Woman})$$

Name	Address
Clark K.	Metropolis
Lois L.	Metropolis

Difference

Man

Name	Address
James B.	London
Clark K.	Metropolis
Tarzan	London

Woman

Name	Address
Lois L.	Metropolis
Catwoman	Gotham City

- What are the cities for which we know at least one woman and no man?

$$\pi_{\text{Address}}(\text{Woman}) - \pi_{\text{Address}}(\text{Man})$$

Address
Gotham City

Formalism for these two operations

Let T_1 and T_2 be 2 tables with **same number of columns** and columns have the **same domain**.

- Formalising **union**

  $T_1 \cup T_2$ is the set containing the rows of T_1 and those of T_2.

  $T_1 \cup T_2 = \{ L \mid L \in T_1 \text{ or } L \in T_2 \}$

- Formalising **difference**

  $T_1 - T_2$ is the set containing the rows of T_1 that are not in T_2.

  $T_1 - T_2 = \{ L \mid L \in T_1 \text{ and } L \notin T_2 \}$

Query Tree

- What are all the possible couples with a man from London and a woman from Metropolis?

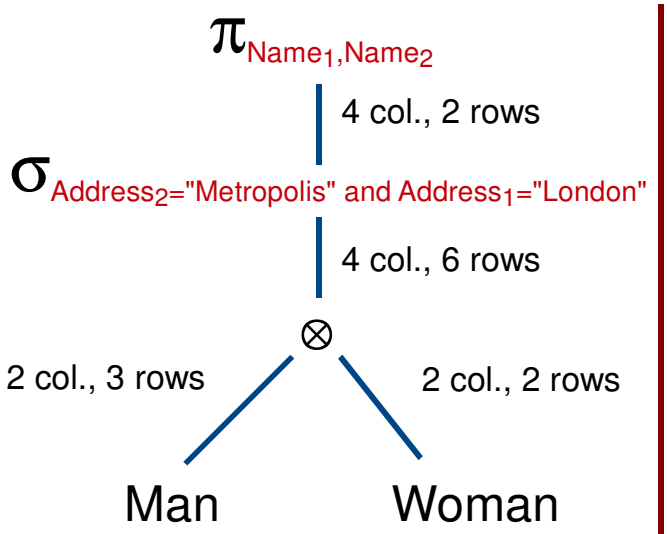
$$\pi_{\text{Name}_1,\text{Name}_2}(\sigma_{\text{Address}_2=\text{"Metropolis"}}(\sigma_{\text{Address}_1=\text{"London"}}(\text{Man} \otimes \text{Woman})))$$

OR

$$\pi_{\text{Name}_1}(\sigma_{\text{Address}_1=\text{"London"}}(\text{Man})) \otimes \pi_{\text{Name}_2}(\sigma_{\text{Address}_2=\text{"Metropolis"}}(\text{Woman}))$$

Query Tree

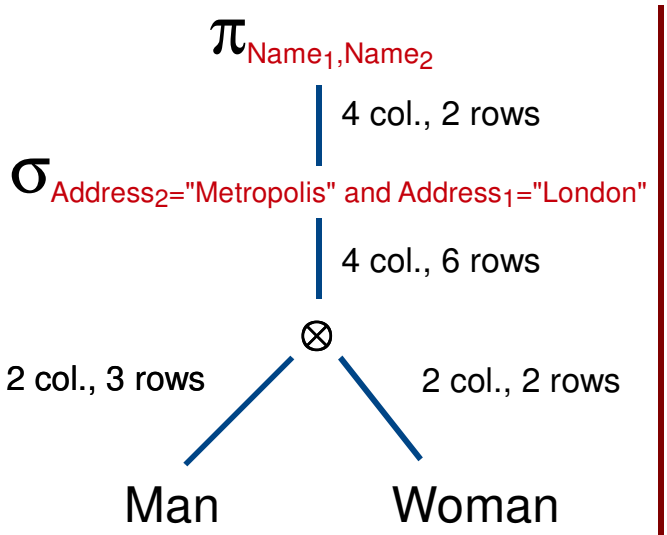
- What are all the possible couples with a man from London and a woman from Metropolis?



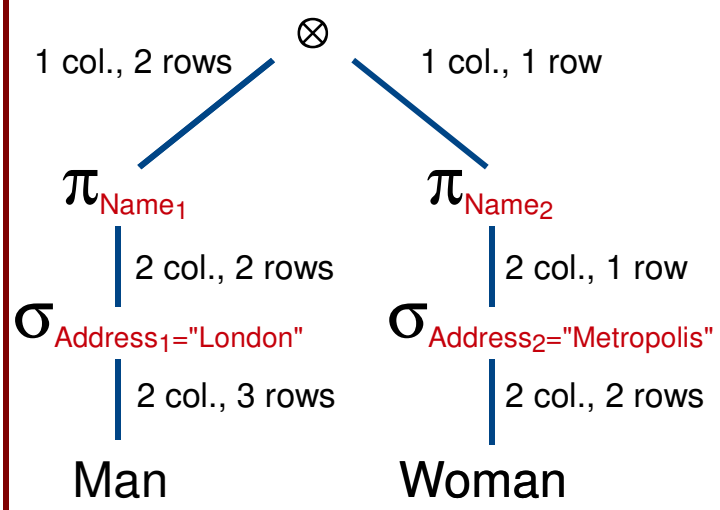
2*3+2*2+4*6+4*2 = **42 cells manipulated**

Query Tree

- What are all the possible couples with a man from London and a woman from Metropolis?



2*3+2*2+4*6+4*2 = **42 cells manipulated**



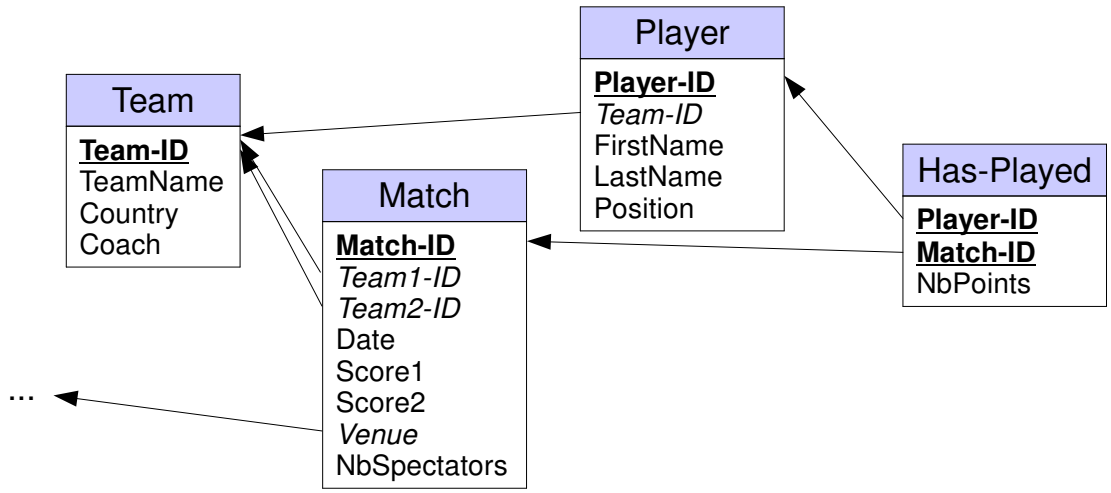
2*3+2*2+2*2+2*1+1*2+1*1 = **19 cells manipulated**

SQL: Structured Query Language

SQL is a database manipulation language developped by IBM (San José, 1981)

- Structured **Query Language**:
  - Can be written by humans
  - Can be saved in files
  - Can be exchanged between applications
- **Structured** Query Language:
  - Can be automatically processed by machines
- SQL is a **standard** but there exist *dialects* (Access, MySQL), i.e., non-standard variants

Ex.: a Rugby Database



Team(**Team-ID**, TeamName, Country, Coach)

Player(**Player-ID**, *Team-ID*, FirstName, LastName, Position)

Match(**Match-ID**, *Team1-ID*, *Team2-ID*, Date, Score1, Score2, *Venue*, NbSpectators)

Has-Played(**Player-ID**, **Match-ID**, NbPoints)

Extension (content of the database)

- Table Team

Team-ID	TeamName	Country	Coach
FRA	XV de France	France	Laporte
NZL	All Blacks	New Zealand	Henry
ENG	XV of the Rose	England	Ashton
AUS	Wallabies	Australia	Connolly
RSA	Spring Boks	South Africa	White
ARG	Pumas	Argentina	Loffreda
SCO		Scotland	Hadden
FJI	Fidgi	Fidgi	Tabua

- Table Match

Match-ID	Team1-ID	Team2-ID	Date	Score1	Score2	Venue	NbSpectators
41	FRA	NZL	06/10/2007	20	18	Cardiff	73350
42	AUS	ENG	06/10/2007	10	12	Marseille	59500
43	RSA	FJI	07/10/2007	37	20	Marseille	59500
44	ARG	SCO	07/10/2007	19	13	Saint Denis	80000
45	ENG	FRA	13/10/2007	14	9	Saint Denis	80000
46	RSA	ARG	14/10/2007	37	13	Saint Denis	80000
47	FRA	ARG	19/10/2007	10	34	Paris	47870
48	RSA	ENG	20/10/2007	15	6	Saint Denis	80000

Extension (contd.)

- Table Player

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

Table Has-Played

Player-ID	Match-ID	NbPoints
1	41	21
1	45	15
1	47	8
2	41	12
2	45	6
2	47	7
3	41	10
6	41	12
7	41	15
8	41	20
9	41	25
17	42	14
17	45	22
17	48	16
18	43	10
18	46	11
18	48	9
19	43	24
19	46	22
19	48	30
...	...	...

Simple Queries

- General form:

```
SELECT <listOfColumns>          -- This is the SELECT clause
FROM <listOfTables>;            -- This is the FROM clause
```

DO NOT FORGET THIS!

Simple Queries

- General form:

```
SELECT <listOfColumns>
FROM <listOfTables>;
```

- *Ex.: list of all players*

```
SELECT *
FROM Player;
```

-- '' means "**all columns**"*

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

Simple Queries

- General form:

```
SELECT <listOfColumns>
FROM <listOfTables>;
```

- *Ex.: list of all players*

```
SELECT *
FROM Player;
```

- In relation algebra:

Player

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

Projection

- Projection on columns: name the columns in the SELECT clause (there can be duplicates)
- *Ex.: list all positions of players*

```
SELECT position
FROM Player;
```

Position
scrumhalf
wing
flyhalf
lock
back row
wing
wing
flyhalf
flyhalf
flyhalf
fullback
wing
wing
...
...
...
...
...
...
...
...

Projection

- Projection on columns: name the columns in the SELECT clause (there can be duplicates)
- *Ex.: list all positions of players*

```
SELECT position
FROM Player;
```

- In relation algebra:

$$\pi_{position}\ (\ Player\ )$$

Position
scrumhalf
wing
flyhalf
lock
back row
wing
wing
flyhalf
flyhalf
flyhalf
fullback
wing
wing
...
...
...
...
...
...
...
...

Projection

- Projection on columns: name the columns in the SELECT clause (there can be duplicates)
- *Ex.: list all positions of players*

```
SELECT position
FROM Player;
```

- In relation algebra:

$$\pi_{position}\ (\ Player\ )$$

- Projection without duplicates

```
SELECT DISTINCT position
FROM Player;
```

Position
scrumhalf
wing
flyhalf
lock
back row
fullback
...

Position
scrumhalf
wing
flyhalf
lock
back row
wing
wing
flyhalf
flyhalf
flyhalf
fullback
wing
wing
...
...
...
...
...
...
...
...

Selection

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <condition>;
```

-- *Projection (the SELECT clause)*

-- *Selection (the **WHERE clause**)*

Selection

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <condition>;
```

-- *Projection (the SELECT clause)*

-- *Selection (the **WHERE clause**)*

- *Ex.: list of French players' names*

```
SELECT FirstName, LastName
FROM Player
WHERE Team-ID = 'FRA';
```

FirstName	LastName
Jean-Baptiste	Elissalde
Vincent	Clerc
Lionel	Beauxis
Lionel	Nallet
Sébastien	Chabal

Selection

```
SELECT <listOfColumns>          -- Projection (the SELECT clause)
FROM <listOfTables>
WHERE <condition>;              -- Selection (the WHERE clause)
```

- *Ex.: list of French players' names*

```
SELECT FirstName, LastName
FROM Player
WHERE Team-ID = 'FRA';
```

FirstName	LastName
Jean-Baptiste	Elissalde
Vincent	Clerc
Lionel	Beauxis
Lionel	Nallet
Sébastien	Chabal

- In relational algebra:

$$\pi_{\text{FirstName,LastName}}(\sigma_{\text{Team-ID='FRA'}}(\text{Player}))$$

Ordering Results

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <condition>
ORDER BY <column> [<order>] [,<column> [<order>]] …
-- <order> is either ASC or DESC (ascending or descending)
```

Ordering Results

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <condition>
ORDER BY <column> [<order>] [,<column> [<order>]] …
-- <order> is either ASC or DESC (ascending or descending)
```

- *Ex.: list of French players' names, alphabetically ordered*

```
SELECT FirstName, LastName
FROM Player
WHERE Team-ID = 'FRA'
ORDER BY LastName ASC;
```

FirstName	LastName
Lionel	Beauxis
Sébastien	Chabal
Vincent	Clerc
Jean-Baptiste	Elissalde
Lionel	Nallet

Combination of Conditions

Boolean connectors: AND, OR, NOT (negation)

Comparison operators: =, <, >, <=, >=, <>

- *Ex.: matches played by the French team*

```
SELECT Match-ID,Team1-ID,Team2-ID,Date,NbSpectators
FROM Match
WHERE Team1-ID = 'FRA' OR Team2-ID = 'FRA';
```

Match-ID	Team1-ID	Team2-ID	Date	NbSpectators
41	FRA	NZL	06/10/2007	7335
45	ENG	FRA	13/10/2007	8000
47	FRA	ARG	19/10/2007	4787

Combination of Conditions

Boolean connectors: AND, OR, NOT (negation)

Comparison operators: =, <, >, <=, >=, <>

- *Ex.: matches played by the French team*

```
SELECT Match-ID,Team1-ID,Team2-ID,Date,NbSpectators
FROM Match
WHERE Team1-ID = 'FRA' OR Team2-ID = 'FRA';
```

Match-ID	Team1-ID	Team2-ID	Date	NbSpectators
41	FRA	NZL	06/10/2007	73350
45	ENG	FRA	13/10/2007	80000
47	FRA	ARG	19/10/2007	47870

- In relational algebra:

$$\pi_{\text{Match-ID,Team1-ID,Team2-ID,Date,NbSpectators}}(\sigma_{\text{Team1-ID='FRA' or Team2-ID='FRA'}}(\text{Match}))$$

Combination of Conditions (contd.)

- *Ex.: List of matches played by the French team and having more than 60000 spectators*

```
SELECT Match-ID,Team1-ID,Team2-ID,Date,NbSpectators
FROM Match
WHERE (Team1-ID = 'FRA' OR Team2-ID = 'FRA')
AND NbSpectators > 60000;
```

Match-ID	Team1-ID	Team2-ID	Date	NbSpectators
41	FRA	NZL	06/10/2007	73350
45	ENG	FRA	13/10/2007	80000

Combination of Conditions (contd.)

- *Ex.: List of matches played by the French team and having more than 60000 spectators*

```
SELECT Match-ID,Team1-ID,Team2-ID,Date,NbSpectators
FROM Match
WHERE (Team1-ID = 'FRA' OR Team2-ID = 'FRA')
AND NbSpectators > 60000;
```

Match-ID	Team1-ID	Team2-ID	Date	NbSpectators
41	FRA	NZL	06/10/2007	73350
45	ENG	FRA	13/10/2007	80000

- In relational algebra:

$$\pi_{\text{Match-ID,Team1-ID,Team2-ID,Date,NbSpectators}} \left(\sigma_{\text{(Team1-ID='FRA' or Team2-ID='FRA') and NbSpectators>60000}}(\text{Match}) \right)$$

Special Conditions

- The "empty" value `NULL` can be tested with a special syntax:

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <columnName> IS NULL;
```

or

```
SELECT <listOfColumns>
FROM <listOfTables>
WHERE <columnName> IS NOT NULL;
```

Do not write:

```
WHERE <columnName> = NULL;
```

Cartesian Product

- Name the tables in the FROM clause

```
SELECT *
FROM Player,Has-Played;
```

First player

Player-ID	Team-ID	FirstName	LastName	Position	Player-ID	Match-ID	NbPoints
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	1	41	21
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	1	45	15
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	1	47	8
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	2	41	12
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	2	45	6
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	2	47	7
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	3	41	10
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	6	41	12
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	7	41	15
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	8	41	20
...	...	...	...	...	...	...	...
2	FRA	Vincent	Clerc	wing	1	41	21
2	FRA	Vincent	Clerc	wing	1	45	15
2	FRA	Vincent	Clerc	wing	1	47	8
2	FRA	Vincent	Clerc	wing	2	41	12

- In relation algebra:

  Player ⊗ Has-Played

Second player

Join

Cartesian product followed by a selection

- *Ex.: List of players who played match #41*

```
SELECT Team-ID,LastName,Match-ID
FROM Player,Has-Played
WHERE Player.Player-ID = Has-Played.Player-ID
AND Match-ID = 41;
```

Team-ID	LastName	Match-ID
FRA	Elissalde	41
FRA	Clerc	41
FRA	Beauxis	41
NZL	Rokocoko	41
NZL	Howlett	41
NZL	Carter	41
NZL	Evans	41

Join

Cartesian product followed by a selection

- *Ex.: List of players who played match #41*

```
SELECT Team-ID,LastName,Match-ID
FROM Player,Has-Played
WHERE Player.Player-ID = Has-Played.Player-ID
AND Match-ID = 41;
```

Team-ID	LastName	Match-ID
FRA	Elissalde	41
FRA	Clerc	41
FRA	Beauxis	41
NZL	Rokocoko	41
NZL	Howlett	41
NZL	Carter	41
NZL	Evans	41

- In relational algebra

$$\pi_{\text{Team-ID,LastName,Match-ID}}(\text{Player} \underset{\text{Player-ID = Player-ID}}{\bowtie} \text{Has-Played})$$

Join of more tables

- *Ex.: Points scored by the All Blacks players*

```
SELECT    LastName,TeamName,Match-ID,Date,
          Team1-ID,Team2-ID,NbPoints
FROM      Match,Player,Has-Played,Team
WHERE     Player.Player-ID = Has-Played.Player-ID
AND       Match.Match-ID = Has-Played.Match-ID
AND       Team.Team-ID = Player.Team-ID
AND       TeamName = 'All Blacks';
```

LastName	TeamName	Match-ID	Date	Team1-ID	Team2-ID	NbPoints
Rokocoko	All Blacks	41	06/10/07	FRA	NZL	12
Howlett	All Blacks	41	06/10/07	FRA	NZL	15
Carter	All Blacks	41	06/10/07	FRA	NZL	20
Evans	All Blacks	41	06/10/07	FRA	NZL	25

Join of more tables

- *Ex.: Points scored by the All Blacks players*

```
SELECT   LastName,TeamName,Match-ID,Date,
         Team1-ID,Team2-ID,NbPoints
FROM     Match,Player,Has-Played,Team
WHERE    Player.Player-ID = Has-Played.Player-ID
AND      Match.Match-ID = Has-Played.Match-ID
AND      Team.Team-ID = Player.Team-ID
AND      TeamName = 'All Blacks';
```

LastName	TeamName	Match-ID	Date	Team1-ID	Team2-ID	NbPoints
Rokocoko	All Blacks	41	06/10/07	FRA	NZL	12
Howlett	All Blacks	41	06/10/07	FRA	NZL	15
Carter	All Blacks	41	06/10/07	FRA	NZL	20
Evans	All Blacks	41	06/10/07	FRA	NZL	25

- In relational algebra

$$\pi_{\text{LastName,TeamName,Match-ID,Date,Team1-ID,Team2-ID,NbPoints}}$$
$$(\sigma_{\text{Player.Player-ID=Has-Played.Player-ID}}(\text{Match}\otimes\text{Player}\otimes\text{Has-Played}\otimes\text{Team})$$
$$\text{and Match.Match-ID=Has-Played.Match-ID and Team.Team-ID=Player.Team-ID and TeamName="All Blacks"}$$

Nested Queries

- By use of set-operators:

```
IN        -- set membership
NOT IN    -- set non-membership
```

`IN` is an operator that determine whether the value of an expression is equal to one or more values belonging to a set.

- *Ex.: List of players who played match #41, with projection onto the player's last name*

```
SELECT LastName
FROM Player
WHERE Player-ID IN
   (SELECT Player-ID
    FROM Has-Played
    WHERE Match-ID = 41);
```

LastName
Elissalde
Clerc
Beauxis
Rokocoko
Howlett
Carter
Evans

Nested Queries (contd.)

`NOT IN` -- *set non-membership*

`NOT IN` is often used instead of the difference operation

- *Ex.: Players who have not played any match at all*

```
SELECT Player-ID
FROM Player
WHERE Player-ID NOT IN
   (SELECT Player-ID
    FROM Has-Played);
```

Player-ID
4
5
10
11
12
13
14
15
16
20
21

Nested Queries (contd.)

`NOT IN` -- set non-membership

`NOT IN` is often used instead of the difference operation

- *Ex.: Players who have not played any match at all*

```
SELECT Player-ID
FROM Player
WHERE Player-ID NOT IN
   (SELECT Player-ID
    FROM Has-Played);
```

Player-ID
4
5
10
11
12
13
14
15
16
20
21

- In relational algebra

$$\pi_{\text{Player-ID}}(\text{Player}) - \pi_{\text{Player-ID}}(\text{Has-Played})$$

Other SQL operations

`UNION` selection of rows that belong to either the first or the second relation (with deletion of duplicates).

- *Ex.: Names of players and coaches of the French team*

```
(SELECT LastName
 FROM Player
 WHERE Team-ID = 'FRA')
UNION
(SELECT Coach
 FROM Team
 WHERE Team-ID = 'FRA');
```

LastName
Elissalde
Clerc
Beauxis
Nallet
Chabal
Laporte

Other SQL operations

`UNION` selection of rows that belong to either the first or the second relation (with deletion of duplicates).

- *Ex.: Names of players and coaches of the French team*

```
(SELECT LastName
 FROM Player
 WHERE Team-ID = 'FRA')
UNION
(SELECT Coach
 FROM Team
 WHERE Team-ID = 'FRA');
```

LastName
Elissalde
Clerc
Beauxis
Nallet
Chabal
Laporte

- Other operators:

`INTERSECTION` selection of rows that belong to both relations

`MINUS` selection of the rows of one relation which do not belong to the second one (difference)

Aggregates

COUNT	Number of values of a set
MAX	Maximum value of a set of values
MIN	Minimum value of a set of values
SUM	Sum of a set of values
AVG	Average of a set of values

- *Ex.: Total number of players*

```
SELECT COUNT(*) AS NbPlayer
FROM Player;
```

NbPlayer
21

Aggregates

COUNT	Number of values of a set
MAX	Maximum value of a set of values
MIN	Minimum value of a set of values
SUM	Sum of a set of values
AVG	Average of a set of values

- *Ex.: Total number of players*

```
SELECT COUNT(*) AS NbPlayer
FROM Player;
```

NbPlayer
21

- *Ex.: Maximum number of spectators*

```
SELECT MAX(NbSpectators) AS MaxAttendance
FROM Match;
```

MaxAttendance
80000

Aggregates (contd.)

COUNT	Number of values of a set
MAX	Maximum value of a set of values
MIN	Minimum value of a set of values
SUM	Sum of a set of values
AVG	Average of a set of values

- *Ex.: Average number of spectators*

```
SELECT AVG(NbSpectators) AS AvgSpectators
FROM Match;
```

AvgSpectators
70027

Or

```
SELECT SUM(NbSpectators) / COUNT(*)
FROM Match;
```

Maths operations (+, -, *, /) can be used

Partitioning relations

GROUP BY — make a **partition** of the result, based on an attribute; aggregates are computed **for each group** in the partition

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

```
SELECT COUNT(*)
AS NbPlayer
FROM Player
GROUP BY Team-ID;
```

Partitioning relations

GROUP BY — make a **partition** of the result, based on an attribute; aggregates are computed **for each group** in the partition

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

```
SELECT COUNT(*)
AS NbPlayer
FROM Player
GROUP BY Team-ID;
```


Partitioning relations

GROUP BY — make a **partition** of the result, based on an attribute; aggregates are computed **for each group** in the partition

Player-ID	Team-ID	FirstName	LastName	Position
1	FRA	Jean-Baptiste	Elissalde	scrumhalf
2	FRA	Vincent	Clerc	wing
3	FRA	Lionel	Beauxis	flyhalf
4	FRA	Lionel	Nallet	lock
5	FRA	Sébastien	Chabal	back row
6	NZL	Joe	Rokocoko	wing
7	NZL	Doug	Howlett	wing
8	NZL	Dan	Carter	flyhalf
9	NZL	Nick	Evans	flyhalf
10	ARG	Felipe	Contepomi	flyhalf
11	ARG	Juan Martin	Hernandez	fullback
12	ARG	Lucas	Borges	wing
13	ARG	Ignacio	Corleto	wing
14	AUS	Matt	Gitean	...
15	AUS	Stirling	Mortlock	...
16	AUS	Drew	Mitchel	...
17	ENG	Jonny	Wilkinson	...
18	RSA	Bryan	Habana	...
19	RSA	Percy	Montgomery	...
20	FJI	Nicky	Little	...
21	SCO	Alasdair	Dickinson	...

```
SELECT COUNT(*)
AS NbPlayer
FROM Player
GROUP BY Team-ID;
```

This is a partition

Partitioning relations

`GROUP BY` make a **partition** of the result, based on an attribute; aggregates are computed **for each group** in the partition

Player-ID	Team-ID	FirstName	LastName	Position	
1	FRA	Jean-Baptiste	Elissalde	scrumhalf	
2	FRA	Vincent	Clerc	wing	
3	FRA	Lionel	Beauxis	flyhalf	5
4	FRA	Lionel	Nallet	lock	
5	FRA	Sébastien	Chabal	back row	
6	NZL	Joe	Rokocoko	wing	
7	NZL	Doug	Howlett	wing	4
8	NZL	Dan	Carter	flyhalf	
9	NZL	Nick	Evans	flyhalf	
10	ARG	Felipe	Contepomi	flyhalf	
11	ARG	Juan Martin	Hernandez	fullback	4
12	ARG	Lucas	Borges	wing	
13	ARG	Ignacio	Corleto	wing	
14	AUS	Matt	Gitean	...	
15	AUS	Stirling	Mortlock	...	3
16	AUS	Drew	Mitchel	...	
17	ENG	Jonny	Wilkinson	...	1
18	RSA	Bryan	Habana	...	2
19	RSA	Percy	Montgomery	...	
20	FJI	Nicky	Little	...	1
21	SCO	Alasdair	Dickinson	...	1

```
SELECT COUNT(*)
AS NbPlayer
FROM Player
GROUP BY Team-ID;
```

NbPlayer
5
4
4
3
1
2
1
1

Partitioning relations (contd.)

`GROUP BY` make a **partition** of the result, based on an attribute; aggregates are computed **for each group** in the partition

- *Ex.: For each player list the sum of scored points*

```
SELECT LastName,SUM(NbPoint) AS Points
FROM Has-Played,Player
WHERE Has-Played.Player-ID = Player.Player-ID
GROUP BY Player-ID;
```

LastName	Points
Elissalde	45
Clerc	25
Beauxis	24
Rokocoko	12
Howlett	15
Carter	20
Evans	25
Wilkinson	52
Habana	30
Montgomery	76

Partitioning relations (contd.)

`HAVING` apply a condition on aggregates

- *Ex.: List total score for players having played more than 2 matches (and order by points scored)*

```
SELECT LastName,SUM(NbPoint) AS Points
FROM Has-Played,Player
WHERE Has-Played.Player-ID = Player.Player-ID
GROUP BY Player-ID
HAVING COUNT(*) > 2
ORDER BY Points;
```

LastName	Points
Elissalde	45
Clerc	25
Wilkinson	52
Habana	30
Montgomery	76

Updating a database

- **Inserting** a row:

```
INSERT INTO <tableName>[(<field1>,<field2>, … )]
VALUES (<value1>,<value2>, … );
```

- *Ex.: Insert new player Paterson*

```
INSERT INTO Player(Player-ID, Team-ID, FirstName,
                   LastName, Position)
VALUES (22,'SCO','Chris','Paterson','wing');
```

Updating a database

- **Inserting** a row:

```
INSERT INTO <tableName>[(<columnName>,<columnName>, … )]
VALUES (<value1>,<value2>, … );
```

- *Ex.: Insert new player Paterson*

```
INSERT INTO Player(Player-ID, Team-ID, FirstName,
                   LastName, Position)
VALUES (22,'SCO','Chris','Paterson','wing');
```

- **Updating** values of fields in an existing row:

```
UPDATE <tableName>
SET <columnName> = <value1>, <columnName> = <value2>, …
WHERE <condition>;
```

- *Ex.: Paterson changed position*

```
UPDATE Player
SET Position = 'fullback'
WHERE Player-ID = 22;
```


Updating a database (contd.)

- **Deleting** a row:

```
DELETE FROM <tableName>
WHERE <condition>;
```

- *Ex.: Chabal retired from rugby*

```
DELETE FROM Player
WHERE Player-ID = 5;
```

Tables Management

- **Creating** tables:

```
CREATE TABLE <tableName>
(<column1> <description>, <column2> <description>, …);
```

- *Ex.: table* Match

```
CREATE TABLE Match
(Match-ID INTEGER PRIMARY KEY,
 Team1-ID TEXT FOREIGN KEY REFERENCES Team(Team-ID),
 Team2-ID TEXT FOREIGN KEY REFERENCES Team(Team-ID),
 Date DATETIME,
 Score1 INTEGER,
 Score2 INTEGER,
 Venue TEXT,
 NbSpectators INTEGER);
```


Tables Management

- **Deleting** tables:

```
DROP <tableName>;
```

- Creating/deleting an **index**, adding **integrity constraints**, manageing **multiple databases**, setting **default** values and much more...

```
CREATE [UNIQUE] INDEX <indexName>
ON <tableName> (<columnName>, … );

DROP INDEX <indexName>;

(column_definition) CONSTRAINT
<constraintName> (some_other_stuff)

( CREATE | DROP ) SCHEMA <schemaName>;

(column_definition) DEFAULT <defaultValue>

VIEW | TRANSACTION | DOMAIN | CHECK | …
```

SQL is a complete programming language!