COURS : BASES DE DONNÉES RELATIONNELLES ET LANGAGE SQL

I) INTRODUCTION

I.1. Bases de données

Depuis toujours, les sociétés humaines ont eu besoin de conserver des informations, comme les archives de l'état civil, les fichiers clients, inventaires, emplois du temps, etc.

Aujourd'hui, la quantité de données à gérer dépasse largement les capacités humaines. Chaque école, entreprise, ou site web manipule des milliers d'enregistrements. Il faut donc un moyen de stocker, organiser et retrouver efficacement ces informations.

Prenons l'exemple d'un festival de musique et imaginons que l'on doive gérer son organisation. Certains artistes participent à des concerts, chaque concert a lieu sur une scène à une date donnée et des spectateurs achètent des billets. Il faut alors pouvoir répondre rapidement à des questions comme « quels artistes jouent sur la scène Kerouac en juillet ? », « Combien de billets ont été vendus pour la date du 14 juillet ? », « Quels sont les concerts d'artistes français ? », etc.

Toutes ces questions nécessitent de retrouver des informations à partir de données structurées. C'est le rôle d'une base de données relationnelle.

I.2. Systèmes de gestion de bases de données (SGBD)

Une base de données est un ensemble structuré de tables. Un **SGBD** (Système de Gestion de Bases de Données) est le logiciel qui permet de stocker les données, de les interroger avec un langage standardisé et de garantir la cohérence et la sécurité des informations.

Il répond à un cahier des charges précis :

- À cause de la très grande taille de certaines bases et du très grand nombre d'accès, le stockage et les requêtes doivent être optimisés pour que les réponses aux requêtes soient les plus rapides possible.
- Le stockage doit être sûr : on ne peut pas se permettre de perdre un dossier médical, un casier judiciaire ou un compte en banque, il faut donc que les données soient stockées plusieurs fois à des endroits différents, si possible sur des supports différents (disques durs, bandes magnétiques...). Le système devra en outre être capable de gérer la reprise sur panne.
- On doit pouvoir interroger et mettre à jour ces données indépendamment de l'endroit où l'on se trouve dans le monde, l'architecture du système de stockage devra donc s'inscrire dans le cadre d'un service réseau.
- Formuler une requête doit être raisonnablement simple (l'utilisateur qui interroge la base ne doit pas avoir de programme à écrire), d'où l'emploi d'un langage standardisé de très haut niveau d'abstraction, le SQL (Structured Query Language), voire d'interfaces graphiques ou GUI (Graphical User Interface) qui permettent d'interagir avec la base avec des compétences réduites en informatique.
- Le système doit être capable de gérer des transactions. Par exemple, dans le cadre d'un transfert de fonds, le compte bancaire créditeur ne doit pas être crédité si le compte débiteur n'a pas été débité ! Dans l'exemple des sites marchands, si plusieurs personnes tentent simultanément d'acheter un objet ou un service, il ne faut pas que l'objet puisse être vendu plusieurs fois (le SGBD doit gérer la concurrence d'accès).

La solution la plus souvent adoptée est celle qui consiste à utiliser une **base de données relationnelle** gérée via un SGBD au sein d'un modèle réseau client/serveur.

On peut citer :
- SQLite (léger, utilisé dans les petits projets et en apprentissage),
- PostgreSQL, MySQL (libres et puissants),
- Oracle, SQL Server (utilisés dans les grandes entreprises).

I.3. Architecture d'un SGBD

Les SGBD sont des logiciels d'une grande complexité, il est donc nécessaire de distinguer plusieurs niveaux d'abstraction :
- Au niveau physique, le SGBD gère la façon dont les données sont stockées sur les disques (mémoires permanentes) et la mémoire RAM (volatile) de manière à assurer les meilleures performances possibles : il y a un compromis à trouver entre les performances des requêtes et les performances de stockage et de mise à jour des données (en gros, plus les performances en lecture sont bonnes, et plus les performances en écriture sont mauvaises). Un utilisateur averti peut aussi intervenir à ce niveau en fonction de besoins spécifiques.
- La base est conçue et gérée au niveau logique. C'est à ce niveau que l'administrateur de la base va gérer l'organisation des tables, les relations et les requêtes.
- Au niveau de l'utilisateur, l'accès aux données est limité à ce qu'on appelle des vues (l'utilisateur ne voit que la partie des données qui l'intéresse), le plus souvent à travers une interface graphique (GUI) délivrée au travers d'une page Web.

Par exemple, sur un site de billetterie, le spectateur ne voit que ses billets et il n'a accès qu'à une vue de la base. Le spectateur n'a aucun pouvoir de modification sur les prix ou les dates des concerts. Tous les utilisateurs interagissent avec la base via la GUI du site Web. Aucun utilisateur ne connaît la façon dont sont réellement stockées les données.

Niveau utilisateur grand public : vues de la base via une GUI
Niveau administrateur : conception et administration via le langage SQL
Niveau développeur du SGBD : gestion physique du stockage

Figure 1 : Architecture d'un SGBD

I.4. Pourquoi utiliser un SGBD ?

En premier lieu, on peut se demander si tout simplement stocker les données dans des séquences (des listes par exemple) avec un langage tel que Python ne suffirait pas ? Premier problème : Python est un langage trop lent. Qu'à cela ne tienne, choisissons un langage rapide tel que le C.

Apparaît un autre problème : stocker les données dans des séquences n'autorise qu'une lecture (parcours de la structure) en O(n), ce qui est trop lent si les données sont volumineuses et les accès très fréquents. Très bien, alors utilisons une structure de données qui permet une variante de recherche dichotomique en log(n) (des arbres spécifiquement conçus).

Encore un problème : quelle que soit la structure de données, elle n'existe que dans la RAM, elle disparaît donc à l'extinction de la machine, ce qui n'est pas acceptable. Écrivons alors les données sur disque dur dans des fichiers :

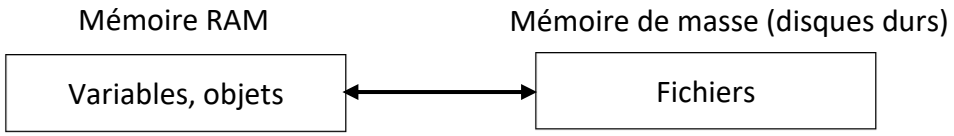


Figure 2 : Mémoire volatile et mémoire permanente.

On pourrait par exemple imaginer stocker les données dans un tableur ou un fichier .csv. D'ailleurs, les structures de données sous-jacentes aux systèmes de fichiers et aux SGBD sont les mêmes (des arbres), et ils partagent certaines fonctionnalités (on peut choisir d'indexer son système de fichiers pour accélérer la recherche d'un document par exemple). Mais les données deviennent redondantes, les recherches sont lentes et se limitent à des cas simples, et les incohérences apparaissent facilement entre les fichiers.

I.5. Stockage des données : structures arborescentes et relationnelles

Prenons l'exemple du stockage des playlists d'un utilisateur d'un site tel que Deezer dans une base de données (appelée `Base` dans l'exemple ci-dessous). À un utilisateur donné correspond plusieurs playlists, à chaque playlist correspond plusieurs titres, etc. :

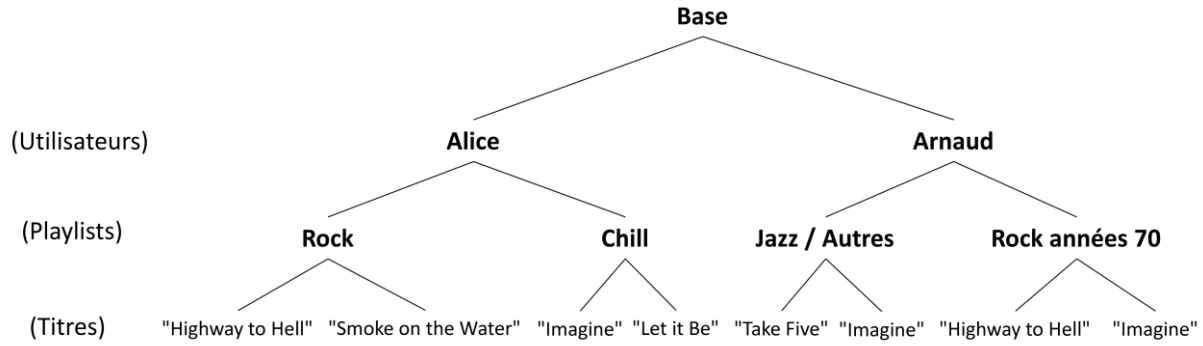


Figure 3 : Structure arborescente des données centrée sur les utilisateurs

Dans cet exemple, on peut remarquer que :

- Les titres « Imagine » et « Highway to Hell » apparaissent plusieurs fois (dans plusieurs branches).
- Si le titre d'un morceau change, il faut le modifier partout.
- Pour savoir dans combien de playlists apparaît un morceau, il faut parcourir tout l'arbre.
- L'organisation est simple par utilisateur, mais pas par morceau.

Cette façon de représenter les données est celle qui préside dans les bases de données non relationnelles utilisées pour le Big Data : chaque portion de l'arbre est autonome (il suffit d'extraire le sous-arbre correspondant à un utilisateur de la base pour avoir toutes les informations qui le concernent), ainsi on évite d'avoir à croiser des données pour interroger la base (ce qui est impossible en pratique quand les données sont trop volumineuses car cela prend trop de temps).

L'inconvénient, c'est qu'on perd énormément de place, car les données sont extrêmement redondantes : on stocke beaucoup de fois le même titre, par exemple. On prend donc le risque que les versions des titres stockées sous le même nom soient différentes. Si cela arrive la base devient alors incohérente : si on interroge la base, on n'est plus assuré d'obtenir un résultat correct.

Un autre inconvénient est qu'il est difficile de visualiser les données car l'arbre peut être très grand. Par exemple, pour répondre à la question « Quels titres sont écoutés par plusieurs utilisateurs ? », il faut parcourir tout l'arbre !

Le **modèle relationnel**, inventé par E. F. Codd (1970), permet d'éviter cela en séparant les informations dans plusieurs tables reliées entre elles par des relations logiques et qui sont interrogeables grâce à au **langage standardisé SQL**.

Contrairement à cette structure arborescente, dans les bases de données relationnelles les données sont **atomisées** afin d'éviter toute duplication. Chaque donnée élémentaire est stockée une seule fois. Elles sont rangées dans des **tables indépendantes**, liées entre elles par des **relations logiques** créées grâce à des « clés étrangères ». Une « clé étrangère » est un « pont » entre deux tables : elle relie une ligne d'une table à la ligne correspondante d'une autre.

Nous reviendrons plus en détails sur le concept de « clé », mais le schéma ci-dessous illustre cette notion :

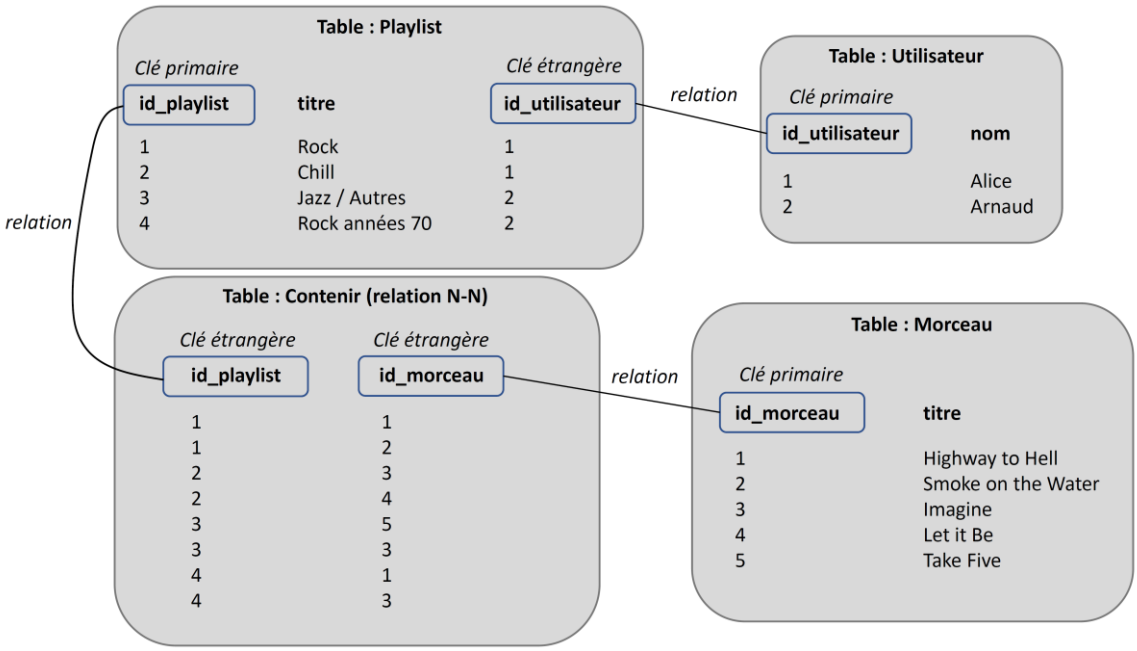


Figure 4 : Structure relationnelle utilisée dans les SGBD

On peut observer que :

- La clé étrangère `id_utilisateur` de la table `Playlist` permet de mettre en relation cette table avec la table `Utilisateur`.
- La clé étrangère `id_morceau` de la table `Contenir` permet de faire la relation avec la table `Morceau`.
- La clé étrangère `id_playlist` de la table Contenir permet de faire la relation avec la table `Playlist`.
- À l'aide de l'ensemble de ces relations, les données stockées ont été atomisées.

Ici, le stockage n'est pas hiérarchisé, toutes les tables se placent au même niveau a priori. On aura la garantie que la base reste dans un état cohérent, parce que chaque donnée ne sera stockée qu'une seule fois (ce qui économisera la mémoire). Le prix à payer sera de devoir croiser les données (voir plus loin la notion de jointure) pour interroger la base : cet aspect est rédhibitoire lorsque la taille des données devient trop grande et contraint alors à abandonner ce modèle qui ne possède quasiment que des avantages.

I.6. Notion de requête

Une requête est une question posée à la base. Elle permet de lire les données stockées, de les filtrer, de les trier, ou de les regrouper.

Il existe deux grandes catégories d'opérations : la lecture des données avec la commande `SELECT` et la modification des données avec les commandes `INSERT`, `UPDATE`, `DELETE`. Dans ce cours, nous ne traiterons que de la lecture des données.

Par exemple, si on souhaite connaître le titre du morceau enregistré dans la table `Morceau` dont `id_morceau` est 2 on peut utiliser la commande SQL :

```
SELECT titre FROM Morceau WHERE id_morceau=3;
```

L'ensemble des attributs qui seront affichés dans le résultat se trouvent après la clause SELECT. L'exemple précédent va donc retourner un résultat suivant :

titre
Imagine

Mais on peut bien entendu demander plusieurs attributs. Par exemple, la commande :

```
SELECT titre, id_utilisateur FROM Playlist WHERE id_playlist=2;
```

Retournera un résultat de la forme :

titre	id_utilisateur
Chill	1

II) LE MODÈLE RELATIONNEL

II.1. Table, attributs et enregistrements

Dans une base relationnelle, les informations sont regroupées dans des **tables** (appelées aussi relations – d'où le nom de bases de données relationnelles). Les lignes de ces tables sont appelées **n-uplets** (tuples en anglais) ou **enregistrements**, et les colonnes sont associées à des **attributs**.

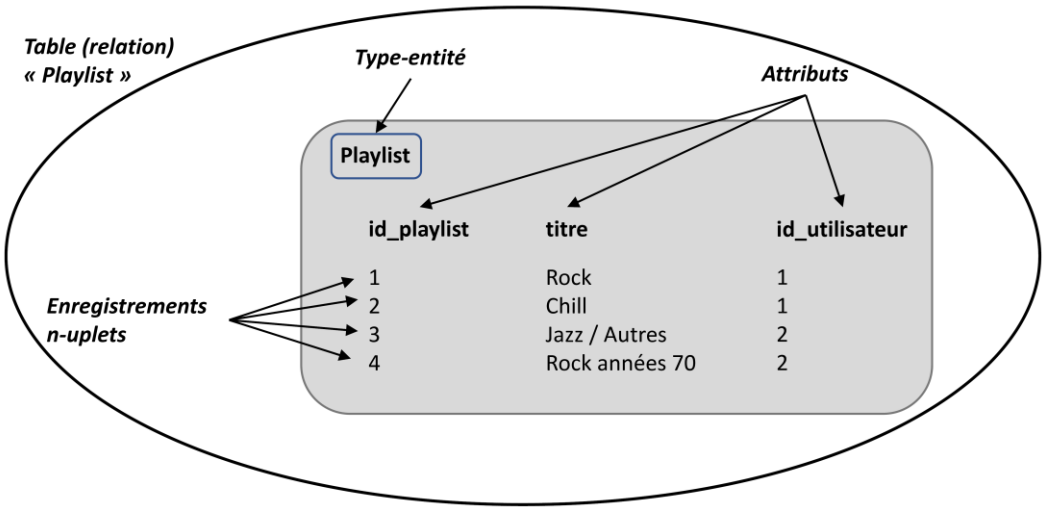


Figure 5 : Exemple d'une table (relation)

Dans une base de données relationnelle, chaque table représente un **type d'entité** du monde réel. Un type-entité correspond à une catégorie d'objets partageant les mêmes caractéristiques : il définit la structure de la table (les colonnes) et la nature des informations qu'elle contient.

Par exemple, la table `Playlist` représente le type-entité « Playlist », défini par des attributs comme `id_playlist`, `titre` ou `id_utilisateur`.

Chaque ligne de la table correspond alors à une entité, c'est-à-dire une occurrence concrète de ce type, une playlist particulière. Ainsi, dans notre exemple, les enregistrements « Rock », « Chill », « Jazz/Autres » et « Rock années 70 » sont autant d'entités du type-entité « Playlist ».

Le **domaine d'un attribut**, fini ou infini, est l'ensemble des valeurs qu'il peut prendre. Le domaine de l'attribut `titre` est une chaine de caractères de taille a priori non connu. Celui de l'attribut `id_utilisateur` est entier.

Le **degré** d'une relation est son nombre d'attributs, 3 dans notre exemple (`id_playlist`, `titre`, `id_utilisateur`).

On représente une table par son nom suivi de ses attributs entre parenthèses. L'ensemble est appelé **schéma de la relation** :

```
Playlist(id_playlist : Entier, titre : Chaîne , id_utilisateur : Entier)
```

On appelle **schéma relationnel** l'ensemble des schémas de relation. On aura évidemment un attribut en commun entre chacune des relations, reliant toutes les données entre elles.

II.2. Règles de stockage

On stocke les données dans les tables en respectant quelques règles :

- On atomise les données autant que possible, de manière à rendre le système de stockage indépendant de la formulation des requêtes (pas de hiérarchie entre données).
- Aucune donnée ne doit être stockée plusieurs fois inutilement : chaque ligne d'une table est unique et on évite au maximum la redondance partielle d'une ligne à l'autre.
- Aucune donnée ne doit résulter d'un calcul qu'on pourrait effectuer au moment de la requête (par exemple si on connaît toutes les notes d'un élève, il est inutile de stocker sa moyenne).

Reprenons par exemple la structure relationnelle de la figure 4 : on pourrait faire l'économie de la table `Contenir` en ajoutant un `id_morceau` dans la table `Playlist` :

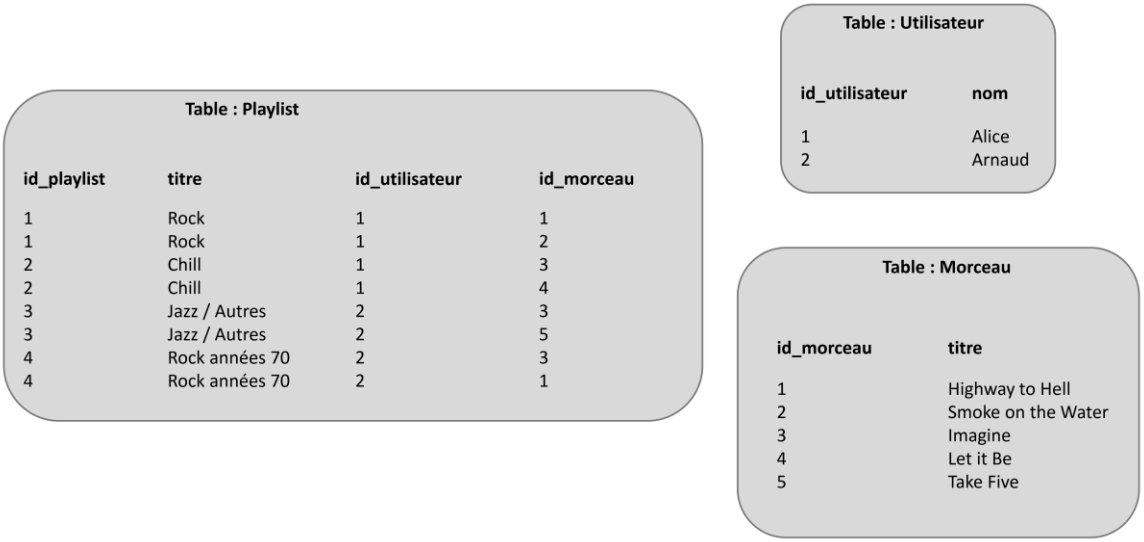


Figure 6 : Structure relationnelle sans la table « Contenir »

Le problème est que dans ce cas on répète plusieurs fois la même playlist (« Rock » apparaît deux fois), on viole la règle fondamentale d'atomisation (une ligne = une entité unique) et la clé primaire `id_playlist` n'est plus unique. On est revenu à un stockage semi-hiérarchique, redondant, fragile, et contraire au principe du modèle relationnel.

En créant un schéma de relation séparé `Contenir(id_playlist, id_morceau)`, on isole la relation elle-même : « Cette playlist contient ce morceau. » :

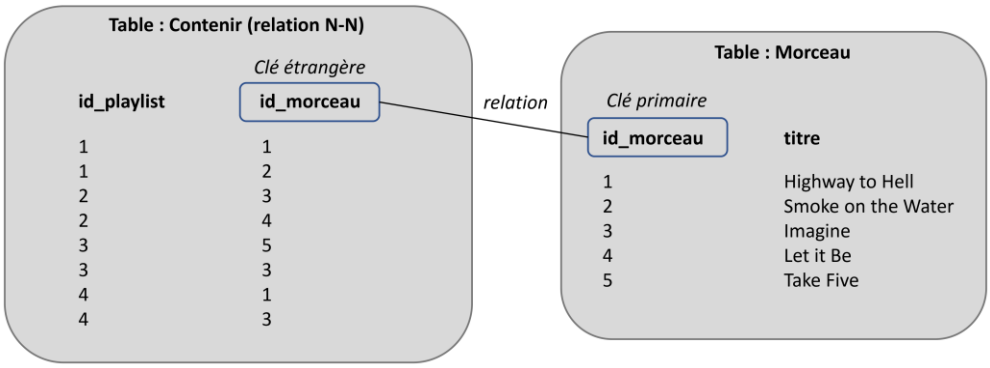


Figure 7 : Relation "Cette playlist contient ce morceau" avec la table Contenir

II.3. Clé primaire d'une table

Il est crucial que chaque enregistrement de chaque table puisse être identifié de façon unique (sinon les réponses aux requêtes seraient ambiguës). C'est pourquoi on définit pour chaque table une **clé primaire**, c'est-à-dire un attribut (ou un ensemble d'attributs) qui permet d'identifier un enregistrement. Il y a une seule clé primaire par table, et dans le modèle relationnel elle existe toujours.

Pour qu'un attribut soit une clé primaire, il faut :

- Qu'il soit obligatoirement renseigné : on dit que la valeur spéciale NULL n'est pas autorisée.
- Que la valeur de ce champ soit unique dans la table.

Conventionnellement, on souligne la clé primaire dans le schéma de relation. Par exemple, si on choisit l'attribut `id_playlist` comme clé primaire de la table `Playlist` :

```
Playlist(id_playlist : Entier, titre : Chaîne , id_utilisateur : Entier)
```

II.4. Clés étrangères

Pour que les tables d'une base soient logiquement reliées entre elles, il faut qu'au moins un attribut d'une table corresponde à un attribut d'une autre table, de manière qu'on puisse ensuite croiser les données provenant de plus d'une table (sinon atomiser les données n'aurait aucun sens). Ce lien est réalisé à l'aide d'une **clé étrangère**. Une clé étrangère sert à **établir une relation entre deux tables**. Elle référence la clé primaire d'une autre table. Elle ne sert pas à identifier une ligne dans sa propre table (donc ce n'est pas une clé primaire).

Par exemple, si l'on considère les deux tables `Playlist` et `Utilisateur`, elles sont logiquement reliées entre elles par l'attribut `id_utilisateur`. Cet attribut est la clé primaire de la table `Utilisateur` et la clé étrangère de la table `Playlist`.

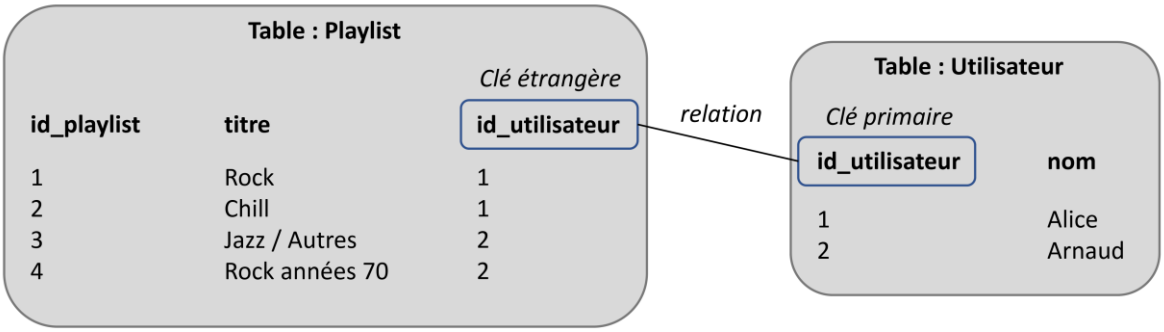


Figure 8 : Exemple de relation entre clé étrangère et clé primaire

Conventionnellement, on souligne en pointillés une clé étrangère dans un schéma de relation :

```
Playlist(id_playlist : Entier, titre : Chaîne , id_utilisateur : Entier)

Utilisateur(id_utilisateur: Entier, nom : Chaîne)
```

Il peut bien entendu y avoir plusieurs clés étrangères dans une table (comme dans la table `Contenir` par exemple).

II.5. Clé primaire composite

Une clé primaire peut être simple (définie par un seul attribut) ou **composite** (définie par plusieurs attributs).

Prenons l'exemple du schéma de relation suivant, dans lequel on a choisi d'utiliser la clé primaire `Nom` dans la table `Etudiant` :

```
Etudiant(Nom, Prenom, Date_naissance, Adresse, Nom_ville)
```

Un tel choix n'est pas judicieux, dans la mesure où il pourrait y avoir deux étudiants qui portent le même nom, ce qui invaliderait l'unicité de la clé primaire. On peut alors choisir de définir le doublet (Nom, Prenom) comme clé primaire. Ce doublet est alors appelé **clé primaire composite** :

```
Etudiant(Nom, Prenom, Date_naissance, Adresse, Nom_ville)
```

Un autre exemple est celui de la table `Contenir`. Dans cette table :

- `id_playlist` est une clé étrangère vers `Playlist(id_playlist)` ;
- `id_morceau` est une clé étrangère vers `Morceau(id_morceau)` ;
- Mais le couple `(id_playlist, id_morceau)` est une clé primaire composite.

En effet, le couple `(id_playlist, id_morceau)` est unique pour chaque association playlist-morceau. Par exemple :

- (1, 1) : le morceau 1 (« Highway to Hell ») est dans la playlist 1 (« Rock ») ;
- (1, 2) : le morceau 2 (« Smoke on the Water ») est dans la playlist 1 (« Rock ») ;
- (3, 3) : le morceau 3 (« Imagine ») est dans la playlist 3 (« Jazz / Autres ») ;
- … ces couples sont tous uniques.

En pratique, pour éviter tout risque de doublon, on définit fréquemment comme clé primaire un attribut additionnel appelé identifiant (il s'agit souvent d'un grand entier). C'est par exemple l'origine des numéros de sécurité sociale, de l'identifiant national étudiant (INE), des plaques d'immatriculation des véhicules, etc.

```
Etudiant(INE , Nom, Prenom, Date_naissance, Adresse, Nom_ville)
```

II.6. Clés secondaires

On peut imaginer d'autres clés que celle déclarée comme clé primaire pour identifier une ligne d'une table (d'autres attributs ou n-uplets d'attributs uniques et toujours renseignés), qui seront alors qualifiées de clés secondaires. Une **clé secondaire** est une clé qui **pourrait servir d'identifiant unique**, mais qui n'a pas été choisie comme clé primaire. On parle aussi de clé candidate, c'est-à-dire une clé potentiellement éligible pour devenir clé primaire.

Par exemple, les deux colonnes `id_utilisateur` et `email` dans la table ci-contre sont uniques pour chaque utilisateur. On pourrait donc choisir l'une ou l'autre comme clé primaire.
Si on choisit `id_utilisateur` comme clé primaire, alors `email` devient une clé secondaire (ou clé candidate).

id_utilisateur	nom	email
1	Alice	alice@mail.com
2	Arnaud	arnaud@mail.com

II.7. Contrainte référentielle

En pratique, dans notre exemple on ne doit pas pouvoir créer une playlist pour un utilisateur qui n'existe pas. Il faut donc faire en sorte qu'une erreur soit affichée si une commande SQL tente d'insérer une playlist pour un utilisateur inexistant.

Cette règle peut être construite à l'aide d'une **contrainte référentielle**. Une contrainte référentielle est une règle imposée par la base de données pour garantir la cohérence entre les tables reliées par une clé étrangère. Cette règle garantit que la clé étrangère correspond toujours à une clé primaire existante.

Dans notre exemple, la contrainte référentielle serait « Chaque valeur de `id_utilisateur` dans `Playlist` doit exister dans la table `Utilisateur` » :

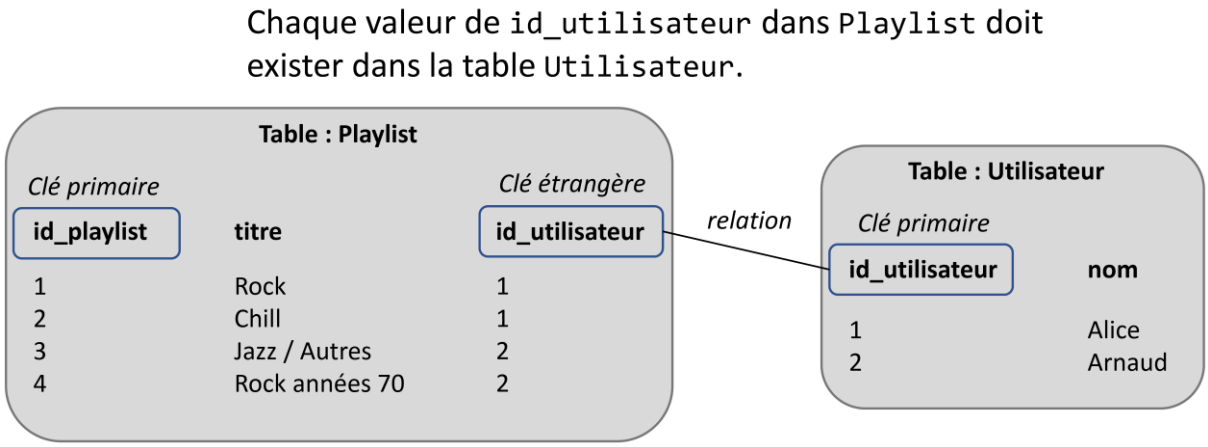


Figure 9 : Contrainte référentielle afin d'éviter l'ajout d'une playlist à un utilisateur inexistant

II.8. Schéma relationnel d'une base

Le graphe formé par l'ensemble des schémas de relations et de leurs contraintes référentielles représentées par des flèches s'appelle le **schéma relationnel de la base**. Sa représentation graphique en 2D n'est pas unique, et on ne peut pas toujours éviter que des flèches se croisent. Il présente l'avantage de visualiser rapidement les liens logiques entre les tables.

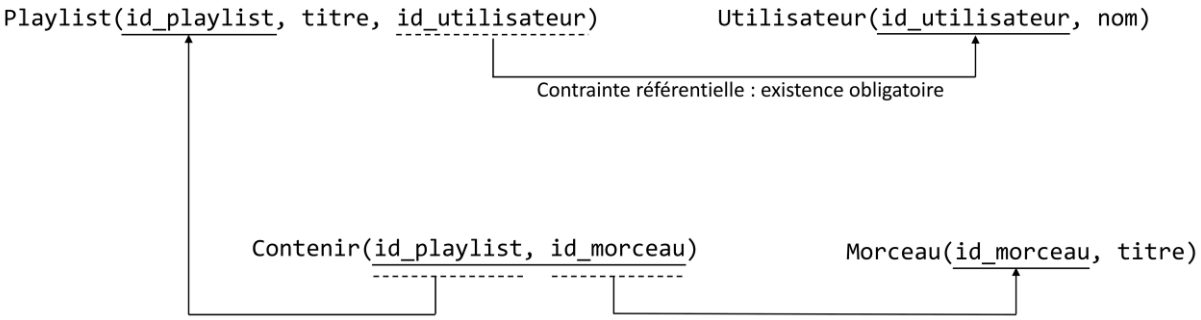


Figure 10 : Exemple de schéma relationnel d'une base

II.9. Contraintes d'intégrité des attributs

Les **contraintes d'intégrité** sont des règles que la base de données impose pour garantir la cohérence et la validité des données. On trouve les types de contraintes suivantes :

- les contraintes de domaine : elles garantissent qu'une valeur individuelle respecte un type ou une condition, par exemple le fait qu'un âge doit être positif ;
- les contraintes d'unicité : elles garantitissent qu'il n'y ait pas deux lignes identiques dans une table. Par exemple, deux utilisateurs ne peuvent pas avoir le même `id_utilisateur` ;
- les contraintes référentielles : elles garantissent que les liens entre tables soient cohérents. Par exemple, on ne peut pas ajouter de playlist à un utilisateur qui n'existe pas ;
- les contraintes de table (ou d'ensemble) : elles garantissent qu'une combinaison de colonnes respecte une condition. Par exemple, une salle ne peut pas accueillir deux cours au même horaire.

Regardons de plus près le cas des contraintes de domaine, que l'on n'a pas encore évoqué, sur l'exemple de la table `Playlist` :

Table : Playlist

id_playlist	titre	id_utilisateur
1	Rock	1
2	Chill	1
3	Jazz / Autres	2
4	Rock années 70	2

Figure 11 : Table Playlist

Cette table comprend ici trois attributs, chaque attribut a un **domaine**.
La notion de domaine contient ici à la fois la notion de type (comme les types des variables en informatique), la notion de domaine de définition propre aux mathématiques et éventuellement la notion de format.

Dans l'exemple de la table `Playlist`, l'attribut `titre` sera défini avec un type `TEXT` et si une commande SQL tente d'insérer une valeur d'un autre type, la base refusera.

Voici quelques exemples de types (la liste complète des types dépend du SGBD) :

- les chaînes de caractères : CHAR(n) pour une chaîne de taille n fixe ou VARCHAR(n) pour une chaîne au plus de taille n. Elles seront notées entre quotes simples, par exemple 'Savigny' ;
- les entiers : INT et ses dérivés (SMALLINT, BIGINT...) ;
- les réels : FLOAT et ses dérivés (DOUBLE...) ;
- les dates : ce sont des chaînes de caractères, mais elles ont en plus un format particulier, par exemple '2023-10-23' (ce format dépend du SGBD).

En ce qui concerne les domaines proprement dits, on peut par exemple imposer :

- un intervalle de valeurs considérées comme correctes (par exemple une date comprise entre deux dates limites) ;
- une liste finie de valeurs imposées (par exemple des valeurs telles que 'M' ou 'Mme'), ce qui est fréquent dans le cas d'un formulaire web ;
- une contrainte de non-nullité : on peut empêcher qu'une colonne soit laissée vide, autrement dit qu'un attribut ait obligatoirement une valeur : par exemple, qu'il soit impossible d'insérer une playlist dans titre ;
- une contrainte de valeur par défaut qui définit une valeur qui sera utilisée si on ne précise rien lors de la création d'un attribut.

III) DU LANGAGE COURANT AU LANGAGE SQL

Le problème qu'on se pose dans cette partie est de transformer une question du langage courant en requête SQL.

On va voir dans la suite qu'il existe trois approches/langages équivalentes qui permettent de passer de l'un à l'autre : le calcul relationnel, l'algèbre relationnelle et le langage SQL.

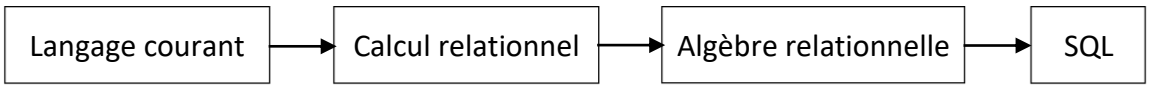


Figure 12 : Les étapes logiques du langage courant à la requête SQL

III.1. Le langage courant et le calcul relationnel

Reprenons notre exemple sur les playlists et supposons qu'on se pose la question suivante : « quel est l'ensemble des titres des playlists de l'utilisateur Alice ? ». On peut tout d'abord traduire la question dans un premier langage appelé **calcul relationnel**.

Le calcul relationnel est une formalisation du langage mathématique inventé par les logiciens de la fin du 19ème siècle basée sur quelques opérateurs logiques élémentaires (qu'on utilise couramment en mathématiques), parmi lesquels :

- $\exists$: il existe
- $\forall$: pour tout
- $\wedge$: et
- $\vee$: ou
- | : tel que
- $\in$: appartient à

La question posée précédemment peut s'exprimer de la manière suivante :

$\{t.titre | t \in Playlist \wedge \exists u \in Utilisateur(u.nom='Alice' \wedge u.id\_utilisateur=t.id\_utilisateur)\}$

…et peut se traduire par :

> *« l'ensemble des attributs `titre` des enregistrements `t` tels qu'ils appartiennent à la table `Playlist` et qu'il existe un utilisateur `u` dans la table `Utilisateur` dont l'attribut `nom` est 'Alice' et l'attribut `id_utilisateur` est `t.id_utilisateur` ».*

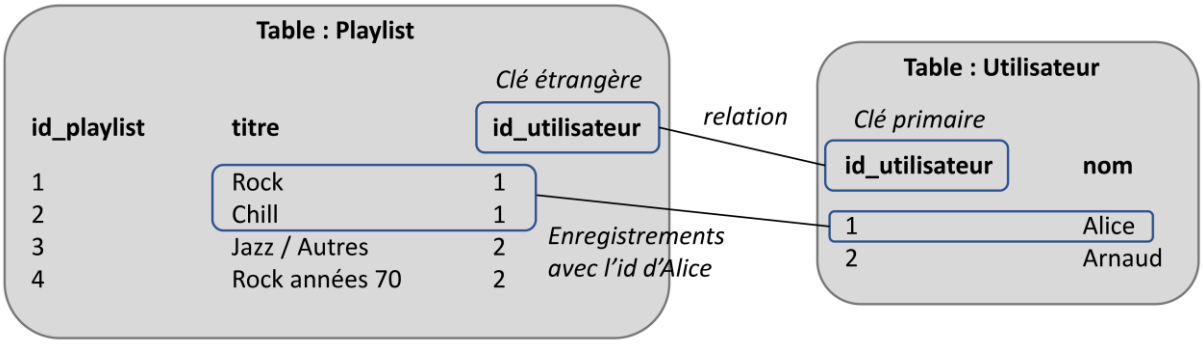


Figure 13 : Relation "quel est l'ensemble des titres des playlists de l'utilisateur Alice "

En plus de la condition `u.nom='Alice'` (condition sur l'utilisateur), on voit ici apparaitre la relation entre la clé primaire `Utilsateur.id_utilisateur` et la clé étrangère `Playlist.id_utilisateur` (`u.id_utilisateur=t.id_utilisateur`). C'est une condition de jointure entre les deux tables.

Il a été démontré par les logiciens que si la question posée a un sens, alors il sera possible de l'exprimer grâce au calcul relationnel. Par exemple une question telle que : quelle est la différence entre le pigeon ? », stupide par nature, ne pourra pas être exprimée par le calcul relationnel.

Même si on peut exprimer la question grâce au calcul relationnel, on n'est pas plus avancé, car le calcul relationnel est un langage déclaratif, c'est-à-dire qu'il va permettre de poser la question, mais sans pour autant nous expliquer comment obtenir la réponse.

III.2. L'algèbre relationnelle

C'est à ce niveau qu'intervient un second langage, l'algèbre relationnelle. Le théorème de Codd (1970) assure que toute question posée via le calcul relationnel pourra être traduite en algèbre relationnelle. On dit que le calcul relationnel et l'algèbre relationnelle ont le même pouvoir d'expression.

L'algèbre relationnelle est un langage procédural, c'est-à-dire qu'il décompose la requête en opérations élémentaires. La conséquence pratique est immédiate : si on réussit à implémenter ces opérations via des fonctions écrites dans un langage de programmation et qu'on peut les composer (au sens de la composition des fonctions mathématiques), alors on a un moyen de répondre à la question.

On distingue deux grandes familles d'opérations supportées par l'algèbre relationnelle :

- les opérations unaires (qui portent sur une seule table) ;
- les opérations binaires (qui combinent deux tables).

Voici quelques exemples d'opérations unaires sur les tables `Contenir` et `Playlist` :

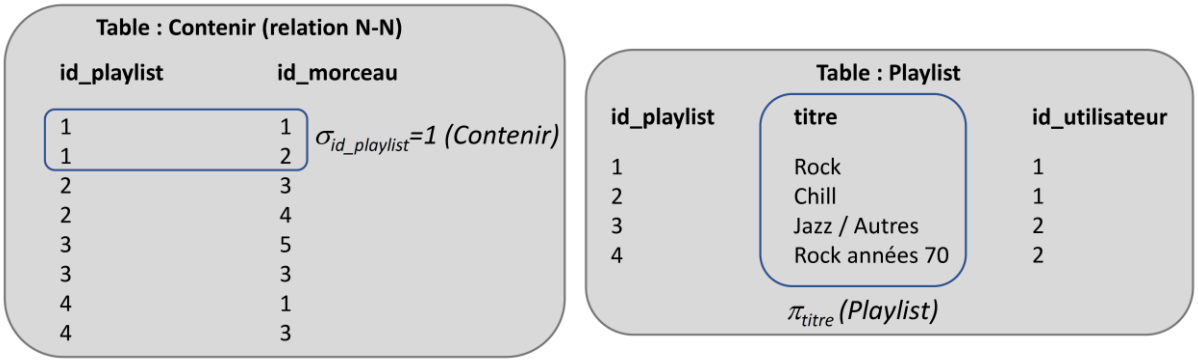


Figure 14 : Tables Contenir et Playlist

Opération	Symbole	Rôle	Exemple
Sélection	σ (sigma)	Garde uniquement les lignes qui vérifient une condition	$\sigma_{id\_playlist} = 1$ (Contenir)
Projection	π (pi)	Garde uniquement certaines colonnes	π_{titre} (Playlist)

Tableau 1 : Exemples d'opérations unaires

Dans les exemples :

- $\sigma_{id\_playlist} = 1$ (Contenir) : Sélectionne les `id_morceau` dont les `id_playlist` sont égaux à 1 dans la table `Contenir`, soit les morceaux appartenant à la playlist 1 ;
- π_{titre} (Playlist) : Garde uniquement la colonne `titre` de la table `Playlist`.

Voici quelques exemples d'opérations binaires :

Opération	Symbole	Rôle	Exemple
Union	∪	réunit les tuples des deux tables (sans doublons)	$R_1 \cup R_2$
Intersection	∩	garde uniquement les tuples communs	$R_1 \cap R_2$
Différence	−	garde les tuples de la 1re table absents de la 2e	$R_1 - R_2$
Produit cartésien	×	associe chaque tuple de R_1 à chaque tuple de R_2	$R_1 \times R_2$
Jointure	$\bowtie_{condition}$	combine les lignes selon une condition commune explicitement exprimée	$R_1 \bowtie_{R_1.A=R_2.B} R_2$ $= \sigma_{R1.A=R2.B}(R1 \times R2)$
Jointure (naturelle)	⋈	jointure sur tous les attributs qui ont le même nom dans les deux relations	$R_1 \bowtie R_2$ $= \sigma_{attributs\_communs}(R1 \times R2)$

Tableau 2 : Exemples d'opérations binaires

Revenons à notre question initiale : « quel est l'ensemble des titres des playlists de l'utilisateur Alice ? ». Elle peut donc être formulée de la manière suivante :

$$\pi_{titre}(\sigma_{nom='Alice'}(Utilisateur \bowtie Playlist))$$

- `Utilisateur` ⋈ `Playlist` : Jointure naturelle – On fait la jointure entre les tables `Utilisateur` et `Playlist`, sur la clé `id_utilisateur`. On combine donc chaque ligne de la table `Playlist` avec les lignes de la table `Utilisateur` qui ont le même `id_utilisateur`.

On récupère donc les données suivantes :

Id_playlist	titre	Id_utilisateur	nom
1	Rock	1	Alice
2	Chill	1	Alice
3	Jazz / Autre	2	Arnaud
4	Rock années 70	2	Arnaud

- $\sigma_{nom='Alice'}$(Utilisateur ⋈ Playlist) : On sélectionne seulement la ligne de la table obtenue précédemment où le nom est Alice. Après cette opération, on récupère donc les données suivantes :

Id_playlist	titre	Id_utilisateur	nom
1	Rock	1	Alice
2	Chill	1	Alice

- $\pi_{titre}(\sigma_{nom='Alice'}(Utilisateur \bowtie Playlist))$: Projection – On ne garde que la colonne `titre`. On obtient donc finalement les données suivantes :

titre
Rock
Chill

Le schéma ci-dessous récapitule l'ensemble des opérations effectuées :

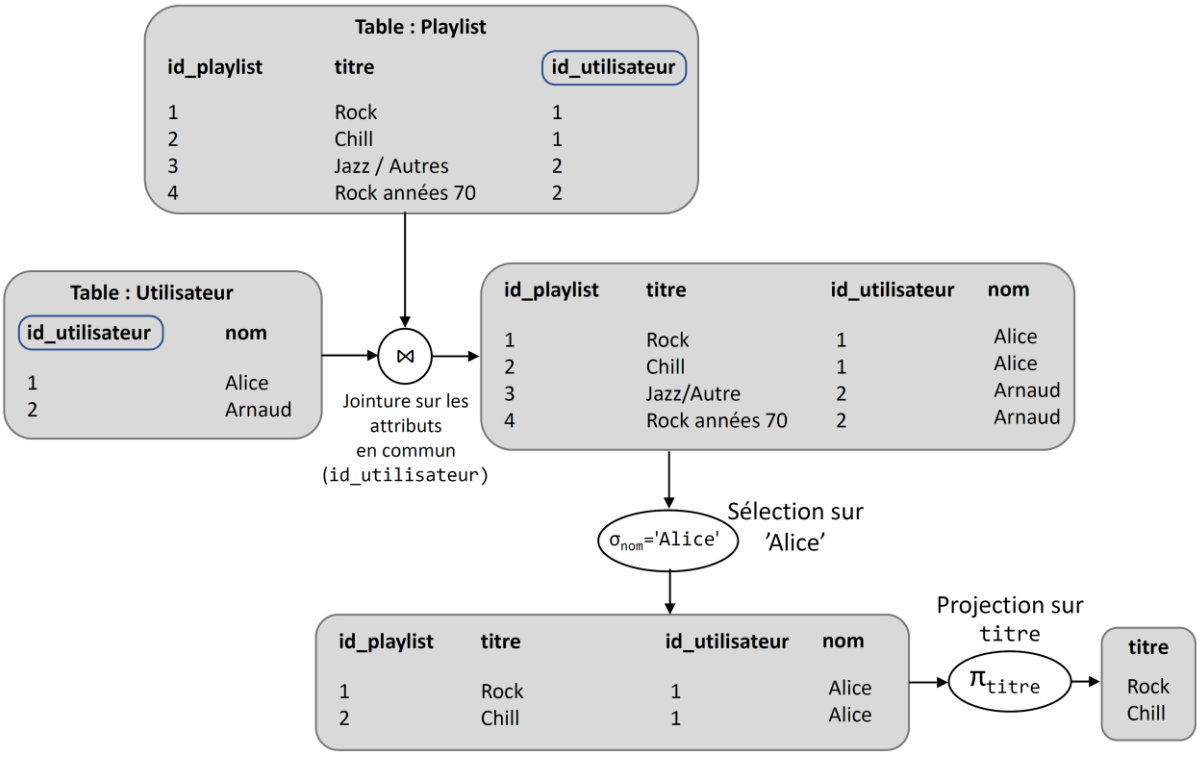


Figure 15 : Opération d'algèbre relationnelle $\pi_{titre}(\sigma_{nom='Alice'}(Utilisateur \bowtie Playlist))$

III.3. Correspondance entre l'algèbre relationnelle et le SQL

Si nous supposons que chaque opération de l'algèbre relationnelle a été implémentée au sein du SGBD, alors il ne reste plus qu'à traduire la requête en langage SQL (qui fera in fine appel aux opérations de l'algèbre).

Le SQL est un langage déclaratif, tourné vers l'utilisateur. Il permet de traduire les requêtes de l'algèbre relationnelle en plus d'offrir des opérations supplémentaires très utiles qui ne font pas partie de l'algèbre (les fonctions d'agrégation, que nous verrons plus loin).

Dans le chapitre suivant, nous allons apprendre à interroger une base de données relationnelle, comprendre la syntaxe et la logique des requêtes SQL et savoir filtrer, trier, regrouper et combiner des données issues de plusieurs tables.

IV) LE LANGAGE SQL : INTERROGER UNE BASE

IV.1. Introduction au langage SQL

Le **langage SQL** (Structured Query Language) est un langage de requêtes structuré. Il permet de lire, interroger et parfois modifier les données d'une base relationnelle.

On distingue trois grands usages :
- création de tables (hors programme ici),
- manipulation des données (`SELECT`, `INSERT`, etc.)
- gestion des droits et transactions (hors programme).

Dans ce cours, on se concentre uniquement sur la manipulation des données, en particulier sur la lecture des données avec la commande `SELECT`.

IV.2. Base de données utilisée comme support dans ce cours

Les bases de données se trouvent sous différentes extensions (csv, sql, xml, xls, bd…). On en trouve par exemple beaucoup concernant la France ici : https://www.data.gouv.fr/fr/

La gestion des bases de données est totalement indépendante du logiciel Python. En effet, le langage SQL peut être directement utilisé dans des logiciels comme « DB Browser for SQLite » gratuit et disponible ici : http://sqlitebrowser.org/. Je vous propose d'utiliser ce logiciel afin de vous exercer et d'entrer les commandes SQL que nous allons étudier au fur et à mesure de votre lecture.

Le fichier de la base de données utilisée est « festival.db ». Cette base de données modélise l'organisation d'un festival de musique regroupant :
- plusieurs artistes de styles différents ;
- des concerts dans plusieurs villes ;
- des spectateurs qui achètent des billets pour y assister et qui ont éventuellement un parrain.

La base de données contient 6 tables liées entre elles :
- `Artiste` : elle contient les informations sur les artistes participant au festival.
- `Identité` : elle contient l'identifiant d'identité unique de l'artiste, et éventuellement son numéro de sécurité sociale.
- `Concert` : elle contient les informations sur chaque concert organisé.
- `Participation` : elle indique quels artistes jouent dans quels concerts et leur cachet. Si un artiste ou un concert est supprimé, ses participations sont automatiquement supprimées ;
- `Spectateur` : contient les informations sur les spectateurs ayant acheté des billets ainsi que leurs parrains éventuels.
- `Billet` : relie chaque spectateur à un concert. Elle représente les billets vendus. Si un concert est supprimé, les billets correspondants disparaissent aussi.

Le schéma relationnel de la base que nous allons utiliser tout au long de cette partie est donné sur la figure 15 :

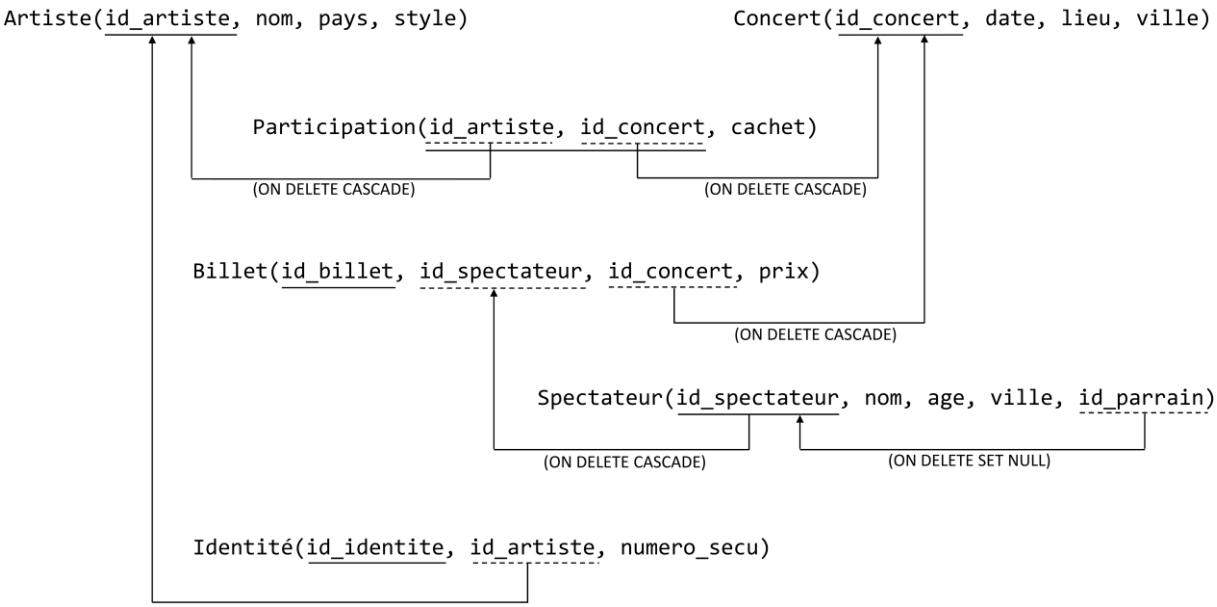


Figure 16 : Schéma relationnel de la base de données "festival.db"

Pour information, mais cela n'est pas à connaître, les termes `ON DELETE CASCADE` et `ON DELETE SET NULL` désignent des types de contraintes référentielles. Elles précisent ce qu'il faut faire si la ligne correspondante dans la table « parente » référencée par la clé étrangère est supprimée.

Par exemple, dans la table `Participation` on a précisé sur la clé étrangère `id_artiste` qui établit une relation avec la table `Artiste` que si on supprime un artiste dans la table `Artiste`, alors toutes ses participations sont automatiquement supprimées. De même, si on supprime un concert dans la table `Concert`, alors toutes les participations à ce concert sont automatiquement supprimées.

D'autre part, la contrainte `ON DELETE SET NULL` sur `id_parrain` signifie que si un parrain est supprimé de la table `Spectateur`, alors la valeur de `id_parrain` de tous ses filleuls est automatiquement remplacée par NULL, au lieu de supprimer les filleuls, ce qui permet de conserver les spectateurs tout en indiquant qu'ils n'ont plus de parrain.

IV.3. Ouverture de la base de données dans le logiciel DB Browser for SQLite

Pour ouvrir le fichier, sélectionner la commande « Ouvrir une base de données …» dans le menu « Fichier », puis ouvrir le fichier « festival.db » :

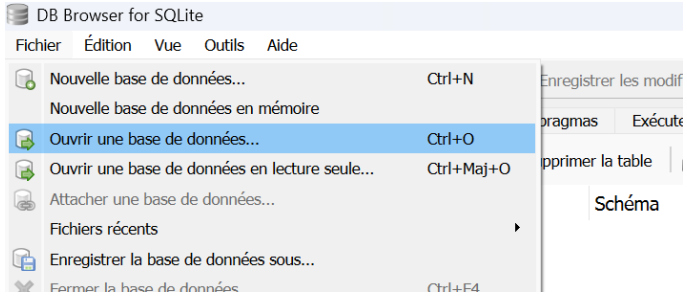


Figure 17 : Ouverture d'une base de données avec DB Browser for SQLite

Cela vous affiche la structure de la base :

Nom	Type	Schéma
Tables (6)		
Artiste		CREATE TABLE Artiste (id_artiste INTEGER PRIMARY KEY, nom TEXT NOT NULL, pays TEXT, style TEXT)
id_artiste	INTEGER	"id_artiste" INTEGER
nom	TEXT	"nom" TEXT NOT NULL
pays	TEXT	"pays" TEXT
style	TEXT	"style" TEXT
Billet		CREATE TABLE Billet (id_billet INTEGER PRIMARY KEY, id_spectateur INTEGER, id_concert INTEGER, prix INTEGER, FOREIGN KEY (id_spectat
id_billet	INTEGER	"id_billet" INTEGER
id_spectateur	INTEGER	"id_spectateur" INTEGER
id_concert	INTEGER	"id_concert" INTEGER
prix	INTEGER	"prix" INTEGER
Concert		CREATE TABLE Concert (id_concert INTEGER PRIMARY KEY, date TEXT, lieu TEXT, ville TEXT)
id_concert	INTEGER	"id_concert" INTEGER
date	TEXT	"date" TEXT
lieu	TEXT	"lieu" TEXT
ville	TEXT	"ville" TEXT
Identite		CREATE TABLE Identite (id_identite INTEGER PRIMARY KEY, id_artiste INTEGER UNIQUE, numero_secu TEXT NOT NULL, FOREIGN KEY (id_ar
id_identite	INTEGER	"id_identite" INTEGER
id_artiste	INTEGER	"id_artiste" INTEGER UNIQUE
numero_secu	TEXT	"numero_secu" TEXT NOT NULL
Participation		CREATE TABLE Participation (id_artiste INTEGER, id_concert INTEGER, cachet INTEGER, PRIMARY KEY (id_artiste, id_concert), FOREIGN KEY
id_artiste	INTEGER	"id_artiste" INTEGER
id_concert	INTEGER	"id_concert" INTEGER
cachet	INTEGER	"cachet" INTEGER
Spectateur		CREATE TABLE Spectateur (id_spectateur INTEGER PRIMARY KEY, nom TEXT NOT NULL, age INTEGER, ville TEXT, id_parrain INTEGER, FORE
id_spectateur	INTEGER	"id_spectateur" INTEGER
nom	TEXT	"nom" TEXT NOT NULL
age	INTEGER	"age" INTEGER
ville	TEXT	"ville" TEXT
id_parrain	INTEGER	"id_parrain" INTEGER
Index (0)		
Vues (0)		
Déclencheurs (0)		

Figure 18 : Schéma de la base "festival.db"

Lorsque vous placez le curseur sur une table (par exemple ci-dessous la table `Billet`), vous obtenez les informations sur les attributs de la table ainsi que leurs types, la clé primaire – `PRIMARY KEY`, la ou les clé(s) étrangère(s) – `FOREIGN KEY` et les éventuelles contraintes référentielles (`ON DELETE CASCADE`) :

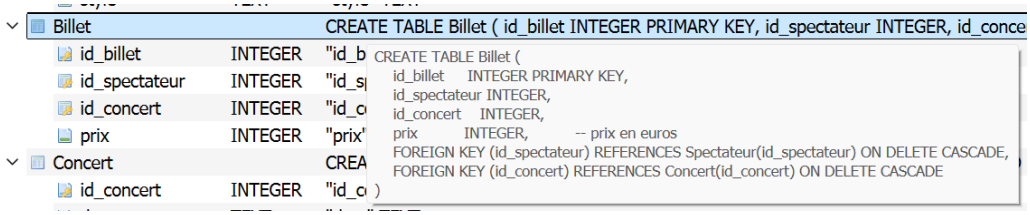


Figure 19 : Informations sur la table Billet

Pour parcourir les données contenues dans une table, il suffit de cliquer sur l'onglet « Parcourir les données » :

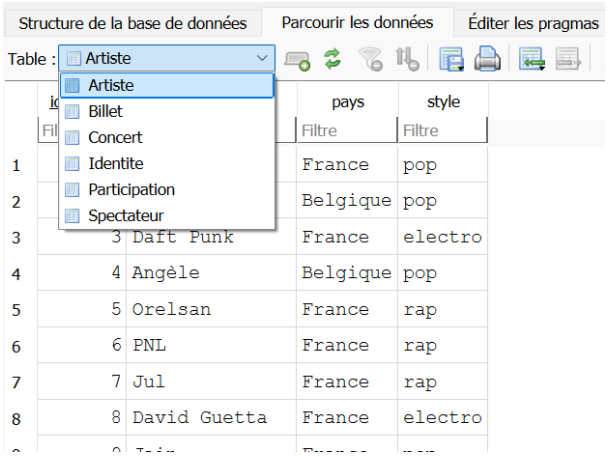


Figure 20 : Contenu de la table Artiste

IV.4. Interroger les bases : la commande SELECT

La commande `SELECT` permet d'interroger une base de données et d'en afficher les informations. C'est l'instruction la plus utilisée en SQL : elle permet de lire, filtrer et trier les données d'une ou plusieurs tables.

IV.4.1. Structure générale de la requête SELECT

La structure générale est la suivante. Les mots entre crochets [] sont facultatifs, mais l'ordre doit toujours être respecté.

```
SELECT [colonnes]
FROM [table]
[WHERE condition]
[ORDER BY colonne]
[LIMIT n];
```

IV.4.2. Exemple simple : lire une table

Pour afficher tout le contenu de la table `Artiste` :

```
SELECT * FROM Artiste;
```

(* signifie « toutes les colonnes »)

IV.4.3. Sélectionner des colonnes précises

Afficher seulement les noms et les styles des artistes :

```
SELECT nom, style FROM Artiste;
```

	nom	style
1	Aya Nakamura	pop
2	Stromae	pop
3	Daft Punk	electro
4	Angèle	pop
5	Orelsan	rap
...	....	...

IV.4.4. Ajouter une condition : WHERE

La clause `WHERE` permet de filtrer les lignes selon un critère.

Exemple : afficher les artistes français.

```
SELECT nom, style FROM Artiste WHERE pays='France';
```

	nom	style
1	Aya Nakamura	pop
2	Daft Punk	electro
3	Orelsan	rap
4	PNL	rap
5	Jul	rap
6	David Guetta	electro
7	Jain	pop

Opérateurs utilisables dans `WHERE` :

Type	Opérateurs	Exemple
Comparaison	=, <>, <, <=, >, >=, BETWEEN	SELECT nom FROM Spectateur WHERE age>20;
Logiques	AND, OR, NOT	SELECT * FROM Artiste WHERE pays='France' AND style='rap';
Texte	LIKE, NOT LIKE	SELECT * FROM Artiste WHERE nom LIKE 'J%';
Liste	IN, NOT IN	SELECT * FROM Artiste WHERE style IN ('pop', 'rap');

Tableau 3 : Opérateurs disponibles dans WHERE

Voici quelques symboles utilisables pour effectuer des recherches particulières :

- « * » permet de dire que l'on recherche tous les attributs
- « % » désigne une chaîne de caractères quelconque – « J% » désignera un mot commençant par J (ou j car LIKE est insensible à la casse ASCII par défaut)
- « _ » désigne un caractère quelconque
- « '…' » désigne un champ au format texte

IV.4.5. L'opérateur BETWEEN

L'opérateur BETWEEN permet de tester qu'une valeur se situe dans un intervalle (inclusif). C'est un moyen plus simple et lisible d'écrire deux comparaisons.

Exemple : Afficher les spectateurs âgés entre 18 et 22 ans :

```
SELECT nom, age FROM Spectateur WHERE age BETWEEN 18 AND 22;
```

Exemple : Afficher les concerts ayant lieu entre le 15 juillet et le 1er août 2025 :

```
SELECT date, ville, lieu FROM Concert
WHERE date BETWEEN '2025-07-15' AND '2025-08-01';
```

	date	ville	lieu
1	2025-07-15	Bordeaux	ParcBordeaux
2	2025-07-16	Marseille	VieuxPort
3	2025-07-17	Lyon	Zenith
4	2025-07-18	Lille	GrandPlace
5	2025-07-19	Paris	Hippodrome
6	2025-08-01	Lyon	Zenith

IV.4.6. Trier les résultats : ORDER BY

Permet d'afficher les résultats dans un certain ordre.

Exemple : trier les artistes alphabétiquement :

```
SELECT nom, style FROM Artiste ORDER BY nom;
```

Exemple : trier les dates des concerts du plus récent au plus ancien :

```
SELECT date, ville FROM Concert ORDER BY date DESC;
```

→

	date	ville
1	2025-08-04	Lille
2	2025-08-03	Marseille
3	2025-08-02	Bordeaux
4	2025-08-01	Lyon
5	2025-07-19	Paris
6	2025-07-18	Lille
7	2025-07-17	Lyon
8	2025-07-16	Marseille
9	2025-07-15	Bordeaux
10	2025-07-14	Paris

`DESC` signifie ordre décroissant (le plus récent en premier)
`ASC` (par défaut) signifie ordre croissant.

IV.4.7. Limiter le nombre de résultats : LIMIT et OFFSET

Permet de n'afficher qu'un certain nombre de lignes.

Exemple : affiche les 5 premiers artistes :

```
SELECT nom, pays FROM Artiste LIMIT 5;
```

→

	nom	pays
1	Aya Nakamura	France
2	Stromae	Belgique
3	Daft Punk	France
4	Angèle	Belgique
5	Orelsan	France

On peut aussi ignorer les premières lignes avec OFFSET. Par exemple, pour sauter les deux premières lignes, puis affiche les trois suivantes :

```
SELECT nom FROM Artiste LIMIT 3 OFFSET 2;
```

→

	nom
1	Daft Punk
2	Angèle
3	Orelsan

IV.4.8. Éliminer les doublons : DISTINCT

On peut ne garder que les valeurs uniques des résultats.

Exemple : afficher la liste des styles de musique sans doublons :

```
SELECT DISTINCT style FROM Artiste;
```

→

	style
1	pop
2	electro
3	rap
4	rock

IV.4.9. Renommer une colonne avec AS

On peut donner un nom temporaire à une colonne dans le résultat. Cela ne change pas la table dans la base, seulement l'en-tête affiché.

```
SELECT nom AS artiste, pays AS origine FROM Artiste;
```

	artiste	origine
1	Aya Nakamura	France
2	Stromae	Belgique
3	Daft Punk	France
4	Angèle	Belgique

On peut également utiliser « AS » afin de simplifier son utilisation ensuite, par exemple :

```
SELECT nom, style FROM Artiste AS A WHERE A.style ='pop';
```

IV.4.10. Exemple complet

Affiche tous les styles musicaux pratiqués par des artistes français, sans doublons, triés par ordre alphabétique :

```
SELECT DISTINCT style AS genre FROM Artiste
WHERE pays = 'France'
ORDER BY genre;
```

→

	genre
1	electro
2	pop
3	rap

IV.4.11. Lien avec l'algèbre relationnelle

L'algèbre relationnelle distingue deux opérations fondamentales :

- Sélection – choisit les lignes qui respectent une condition : $\sigma_{ville='Paris'}(Concert)$
- Projection – choisit les colonnes à afficher : $\pi_{nom,style}(Artiste)$

En SQL, la commande :

```
SELECT nom, style FROM Artiste WHERE pays = 'France';
```

… fait deux choses à la fois :

Partie SQL	Opération relationnelle correspondante	Description
`WHERE pays='France'`	$\sigma_{pays='France'}$	Sélectionne les lignes où le pays est « France »
`SELECT nom, style`	$\pi_{nom,style}$	Projette (affiche) seulement les colonnes nom et style.

Tableau 4 : Équivalence SQL <=> opérations relationnelles

Donc le mot-clé WHERE correspond à l'opérateur de sélection (σ) et le mot-clé SELECT correspond à l'opérateur de projection (π) :

```
SELECT nom, style
FROM Artiste
WHERE pays='France';
```

$\Leftrightarrow$ $\pi_{nom,style}(\sigma_{pays='France'}(Artiste))$

IV.4.12. Synthèse de la commande SELECT

Rôle : La commande **SELECT** permet d'interroger une base de données et d'en afficher les informations. Elle permet de lire, filtrer et trier les données sans les modifier.

Clause	Rôle	Exemple
SELECT	colonnes à afficher (projection)	SELECT nom, style
FROM	table interrogée	FROM Artiste
WHERE	filtrer les lignes (sélection)	WHERE pays='France'
BETWEEN	valeur comprise entre deux bornes	WHERE age BETWEEN 18 AND 22
ORDER BY	trier le résultat	ORDER BY nom DESC
LIMIT / OFFSET	limiter les lignes affichées	LIMIT 5 OFFSET 2
DISTINCT	éviter les doublons	SELECT DISTINCT style
AS	renommer une colonne	SELECT nom AS artiste

Principaux opérateurs de WHERE :

Type	Opérateurs	Exemple
Comparaison	=, <>, <, <=, >, >=, BETWEEN	SELECT nom FROM Spectateur WHERE age>20;
Logiques	AND, OR, NOT	SELECT * FROM Artiste WHERE pays='France' AND style='rap';
Texte	LIKE, NOT LIKE	SELECT * FROM Artiste WHERE nom LIKE 'A%';
Liste	IN, NOT IN	SELECT * FROM Artiste WHERE style IN ('pop', 'rap');

Correspondance avec l'algèbre relationnelle :

Opération relationnelle	Symbole	Équivalent SQL
Projection (colonnes)	π	SELECT
Sélection (lignes)	σ	WHERE

IV.5. Combiner les tables : les jointures (`JOIN…ON…`)

Les jointures permettent de rassembler des données provenant de plusieurs tables en une seule requête. Elles exploitent les relations logiques entre les tables (souvent via des clés primaires et étrangères).

IV.5.1. Jointure sans condition

La **jointure** (sans condition) réalise le produit cartésien entre deux tables. Par exemple, la jointure entre la table `Concert` et la table `Spectateur` va créer une nouvelle table (`Concert JOIN Spectateur`) qui associera chaque ligne de la table `Concert` à chaque ligne de la table `Spectateur`.

Voici ce que donne la commande : `SELECT * FROM Concert JOIN Spectateur;`

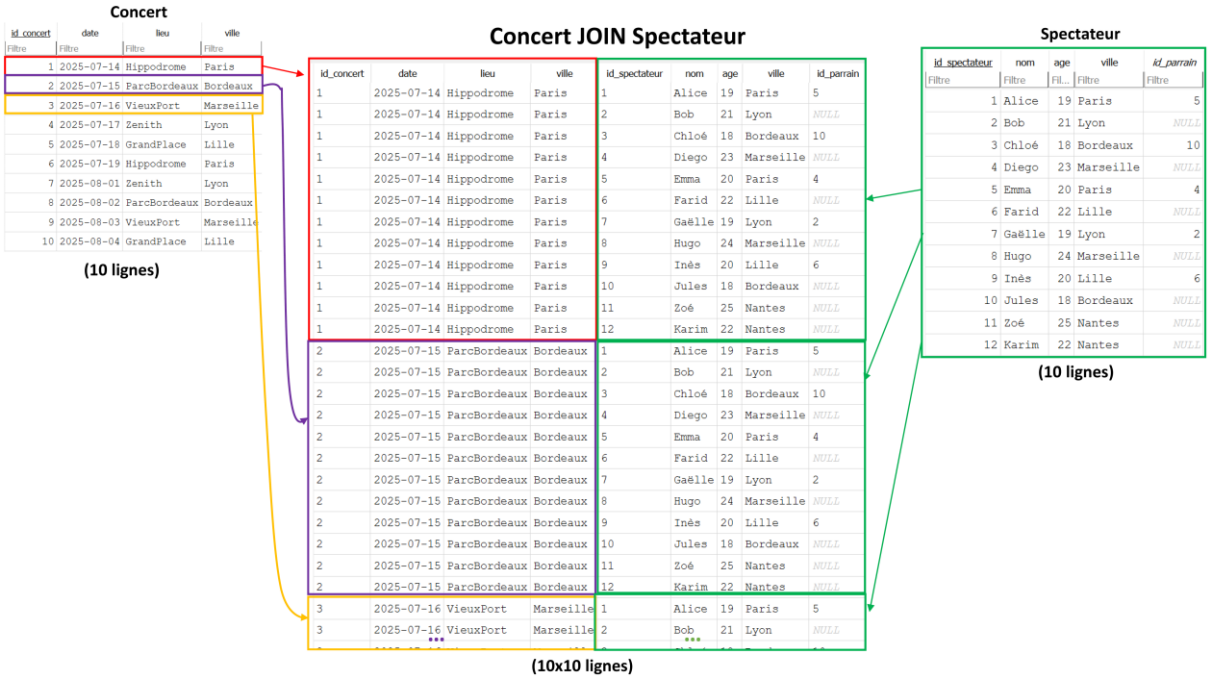


Figure 21 : Exemple d'une opération de jointure (sans condition)

Ici, aucune condition n'a été imposée sur la jointure. On obtiendrait le même résultat avec la commande : `SELECT * FROM Concert CROSS JOIN Spectateur;` (c'est d'ailleurs cette version est recommandée lorsqu'on souhaite faire des jointures sans condition).

Dans la pratique, on n'utilise presque jamais une jointure sans condition, car sans condition, on obtient un produit cartésien, c'est-à-dire toutes les combinaisons possibles entre les lignes des deux tables, ce qui est inexploitable dans la quasi-totalité des cas.

Une jointure sans condition, c'est comme si on mélangeait toutes les lignes au hasard. Pour obtenir des résultats pertinents, on précise toujours une condition de correspondance qui peut être construite grâce à une égalité entre les attributs des tables (on parle dans ce cas de **jointure logique**) ou entre les clés primaires et secondaires des tables (on parle dans ce cas de **jointures relationnelles**).

Voyons maintenant comment on construit ces types de condition sur les jointures.

IV.5.2. Jointure logique (avec une condition sur des attributs)

Dans la pratique, on crée une jointure dans le but d'extraire des résultats pertinents. On impose donc un filtre (une condition) sur la jointure.

Imaginons par exemple qu'on l'on veuille récupérer l'ensemble des concerts qui se déroule dans la même ville que celle où habite le spectateur. Nous pouvons commencer par mélanger les données de la table `Concert` avec celles de la table `Spectateur` (comme nous l'avons fait précédemment) avec la commande :

```
SELECT * FROM Concert JOIN Spectateur;
```

… et faire en sorte de ne mélanger que les lignes de la table `Concert` avec les lignes de la table `Spectateur` pour lesquelles les villes sont identiques, c'est-à-dire les lignes pour lesquelles on a l'égalité sur les attributs `ville` de chaque table, soit les lignes où on a l'égalité `Concert.ville = Spectateur.ville` qui est vérifiée.

C'est ce qu'on appelle faire une **jointure logique** (ici l'attribut `ville` n'est ni une clé primaire ni une clé secondaire, on fait une jointure sur des attributs). Pour ajouter cette condition à l'opération de jointure, on utilise le mot clé `ON` :

```
SELECT * FROM Concert JOIN Spectateur ON Concert.ville=Spectateur.ville;
```

id_concert	date	lieu	ville	id_spectateur	nom	age	ville	id_parrain
1	2025-07-14	Hippodrome	Paris	1	Alice	19	Paris	5
1	2025-07-14	Hippodrome	Paris	5	Emma	20	Paris	4
2	2025-07-15	ParcBordeaux	Bordeaux	3	Chloé	18	Bordeaux	10
2	2025-07-15	ParcBordeaux	Bordeaux	10	Jules	18	Bordeaux	*NULL*
3	2025-07-16	VieuxPort	Marseille	4	Diego	23	Marseille	*NULL*
3	2025-07-16	VieuxPort	Marseille	8	Hugo	24	Marseille	*NULL*
4	2025-07-17	Zenith	Lyon	2	Bob	21	Lyon	*NULL*
4	2025-07-17	Zenith	Lyon	7	Gaëlle	19	Lyon	2
5	2025-07-18	GrandPlace	Lille	6	Farid	22	Lille	*NULL*
5	2025-07-18	GrandPlace	Lille	9	Inès	20	Lille	6
6	2025-07-19	Hippodrome	Paris	1	Alice	19	Paris	5
6	2025-07-19	Hippodrome	Paris	5	Emma	20	Paris	4
7	2025-08-01	Zenith	Lyon	2	Bob	21	Lyon	*NULL*
7	2025-08-01	Zenith	Lyon	7	Gaëlle	19	Lyon	2
8	2025-08-02	ParcBordeaux	Bordeaux	3	Chloé	18	Bordeaux	10
8	2025-08-02	ParcBordeaux	Bordeaux	10	Jules	18	Bordeaux	*NULL*
9	2025-08-03	VieuxPort	Marseille	4	Diego	23	Marseille	*NULL*
9	2025-08-03	VieuxPort	Marseille	8	Hugo	24	Marseille	*NULL*
10	2025-08-04	GrandPlace	Lille	6	Farid	22	Lille	*NULL*
10	2025-08-04	GrandPlace	Lille	9	Inès	20	Lille	6

Figure 22 : Opération de jointure logique (JOIN...ON...)

Une jointure logique (sur les attributs) a plusieurs inconvénients :

- Elle n'assure pas que les valeurs soient correctes ou cohérentes : « Paris » n'est pas la même chose que « paris » ou encore que « PARIS » et il y des risques d'erreurs, de doublons, ou de combinaisons absurdes.
- Elles sont plus lentes que les jointures relationnelles : les colonnes descriptives (ville, nom, etc.) ne sont pas indexées par défaut et SQL doit parcourir toutes les lignes pour comparer les valeurs. Le temps d'exécution est donc plus long.
- Le sens sémantique est flou : ce type de jointure n'exprime pas une vraie relation entre entités. C'est juste une coïncidence de valeurs, il n'y a pas de vrai lien structurel.

IV.5.3. Jointure relationnelle (avec une condition sur les clés)

Contrairement aux jointures logiques, les **jointures relationnelles** sont basées sur les clés et ont des avantages majeurs par rapport aux jointures logiques :

- Cohérence logique et intégrité des données : les clés étrangères garantissent que les valeurs correspondent réellement (par exemple, chaque `Participation.id_artiste` existe dans `Artiste.id_artiste`.) et si un artiste est supprimé, ses participations peuvent être supprimées automatiquement grâce à des contraintes relationnelles (`ON DELETE CASCADE`). Tout cela évite des incohérences.
- Lisibilité et maintenance du code : les jointures sur clés sont standardisées, elles suivent les relations du schéma. Elles sont donc faciles à comprendre et à maintenir, un développeur sait tout de suite ce que relie la jointure.
- Les jointures sur clés sont plus rapides : les clés primaires et étrangères sont généralement indexées (les index sont souvent des arbres B+ ou des tables de hachage), ce qui accélère la recherche de correspondances. Le moteur SQL reconnaît ces relations et peut optimiser les plans d'exécution et comme le SGBD sait qu'une clé primaire est unique, il évite des comparaisons inutiles.

La structure type d'une jointure relationnelle est la suivante :

```
SELECT colonnes
FROM table1
JOIN table2 ON table1.colonne_commune = table2.colonne_commune;
```

… où `colonne_commune` sont des clés (en général clé primaire sur la `table1` et clé étrangère sur la `table2`, ou l'inverse – mais d'autres cas sont possibles comme le cas où les deux clés sont des clés étrangères ou deux clés primaires).

Par exemple, si on souhaite afficher le nom de chaque artiste et son cachet pour chaque concert qu'il a fait, la requête à utiliser est la suivante :

```
SELECT Artiste.nom, Participation.id_concert, Participation.cachet
FROM Artiste
JOIN Participation ON Artiste.id_artiste=Participation.id_artiste;
```

IV.5.4. Alias de tables

L'utilisation d'**alias** permet de rendre les requêtes plus lisibles. Par exemple, la commande précédente :

```
SELECT Artiste.nom, Participation.id_concert, Participation.cachet
FROM Artiste
JOIN Participation ON Artiste.id_artiste=Participation.id_artiste;
```

Peut être remplacée par :

```
SELECT a.nom, p.id_concert, p.cachet
FROM Artiste AS a
JOIN Participation AS p ON a.id_artiste=p.id_artiste;
```

Ici, on a utilisé les alias a et p qui sont des abréviations des tables `Artiste` et `Participation`.

Remarque : On peut également exécuter la commande sans le mot clé AS. Le moteur SQL reconnait l'utilisation des alias :

```
SELECT a.nom, p.id_concert, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste=p.id_artiste;
```

IV.5.5. Jointure sur trois tables

Exemple : Afficher le nom de l'artiste, la ville du concert et le cachet correspondant.

```
SELECT a.nom, c.ville, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON c.id_concert = p.id_concert;
```

	nom	ville	cachet
1	Aya Nakamura	Paris	30000
2	Stromae	Paris	45000
3	David Guetta	Paris	70000
4	Coldplay	Bordeaux	80000
5	Jain	Bordeaux	25000
6	Jul	Marseille	40000
	...	...	

IV.5.6. Trier (`ORDER BY`), filtrer (`WHERE`) et limiter (`LIMIT`) les résultats

On peut rajouter les clauses ORDER BY, WHERE et LIMIT après les jointures.

Actions	Clause	Exemple
Trier	ORDER BY	`…ORDER BY c.ville, a.nom;`
Filtrer	WHERE	`…WHERE c.ville='Paris'`
Limiter	LIMIT	`…LIMIT 5;`

Voici un exemple complet :

```
SELECT a.nom, c.ville, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON c.id_concert = p.id_concert
WHERE c.ville = 'Paris'
ORDER BY p.cachet DESC;
```

	nom	ville	cachet
1	David Guetta	Paris	70000
2	Orelsan	Paris	52000
3	Stromae	Paris	45000
4	Angèle	Paris	42000
5	Aya Nakamura	Paris	30000
6	Jain	Paris	26000

IV.5.7. Utilisation des opérateurs logiques avec les jointures

Une jointure peut relier plusieurs tables, et on peut combiner les conditions avec les opérateurs AND et OR pour affiner le résultat. Les conditions dans ON décrivent le lien entre les tables, celles dans WHERE filtrent les lignes après coup.

Essayons par exemple d'afficher les artistes français ou belges ayant participés à des concerts à Paris ou à Lyon, et dont le cachet est supérieur à 30 000EUR. On doit donc relier trois tables : `Artiste`, `Participation` et `Concert` et appliquer plusieurs conditions logiques sur la jointure et les filtres.

La première étape consiste à joindre ces trois bases selon leurs relations sur les clés primaires et étrangères et à sélectionner les données dont nous avons besoin : le nom de l'artiste, son pays d'origine, la ville où se déroule le concert et le cachet de l'artiste :

```
SELECT a.nom, a.pays, c.ville, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON p.id_concert = c.id_concert;
```

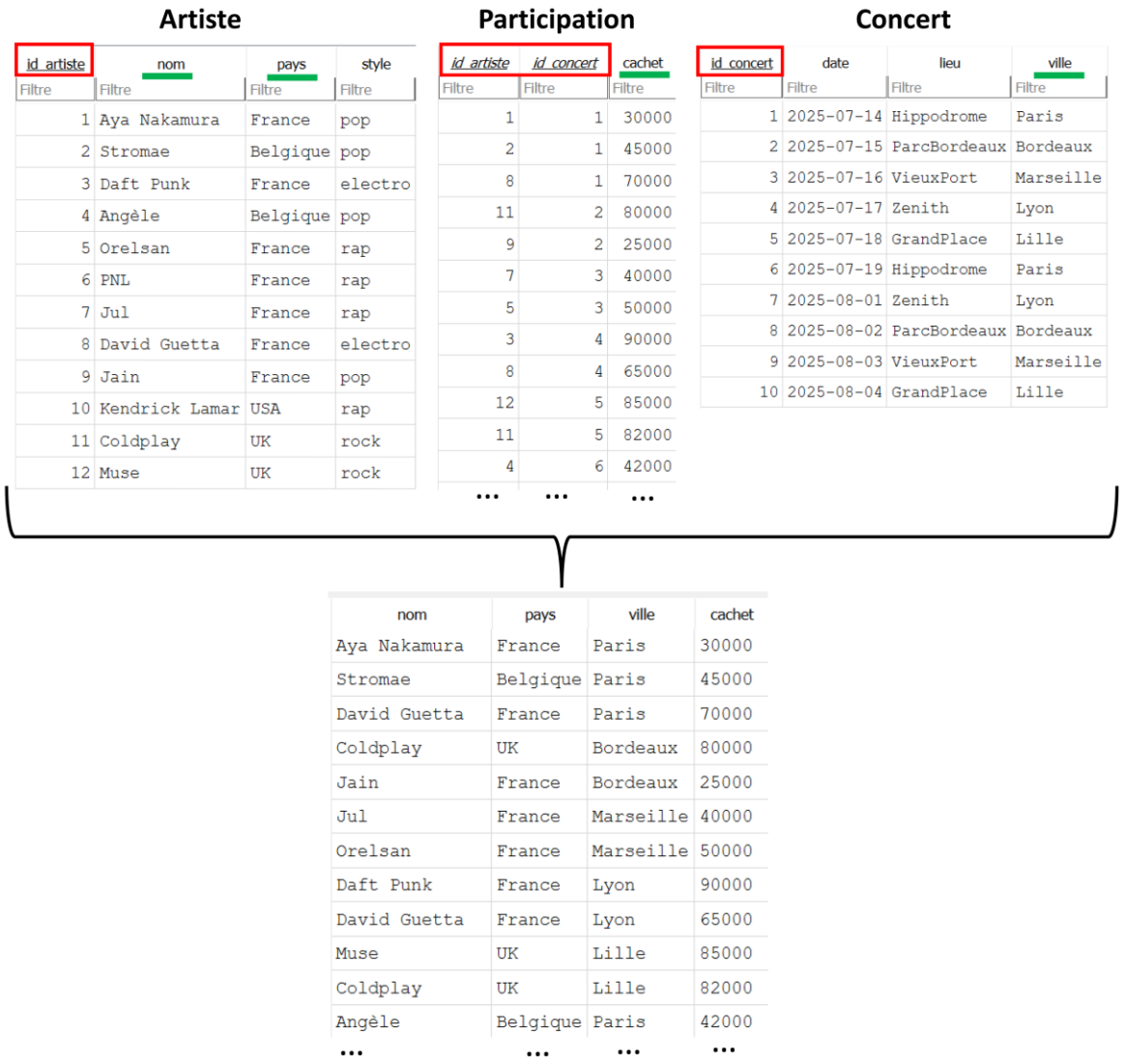


Figure 23 : Jointure des trois tables

Puis on filtre les résultats avec WHERE en y ajoutant les conditions logiques nécessaires :

```
SELECT a.nom, a.pays, c.ville, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON p.id_concert = c.id_concert
WHERE (a.pays = 'France' OR a.pays = 'Belgique')
  AND (c.ville = 'Paris' OR c.ville = 'Lyon')
  AND p.cachet > 30000;
```

	nom	pays	ville	cachet
1	Stromae	Belgique	Paris	45000
2	David Guetta	France	Paris	70000
3	Daft Punk	France	Lyon	90000
4	David Guetta	France	Lyon	65000
5	Angèle	Belgique	Paris	42000
6	Orelsan	France	Paris	52000
7	Orelsan	France	Lyon	60000

ON et WHERE fonctionnent indépendamment l'un de l'autre. Une autre solution serait d'introduire une condition logique directement dans la clause ON si elle concerne le lien entre les tables :

```
SELECT a.nom, a.pays, c.ville, p.cachet
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
AND (a.pays='France' OR a.pays='Belgique')
JOIN Concert c ON p.id_concert = c.id_concert
AND (c.ville='Paris' OR c.ville='Lyon')
AND p.cachet>30000;
```

…mais on préfèrera utiliser ON pour la condition de jointure créant une table « logique », puis WHERE pour restreindre les résultats.

IV.5.8. Les auto-jointures

Une auto-jointure consiste à relier une table avec elle-même. On l'utilise quand les données d'une même table ont un lien logique entre elles. Par exemple :

- Une personne et son supérieur hiérarchique (même table Employé),
- Un spectateur et ses amis dans la même ville,
- Un spectateur et son parrain,
- Un concert et d'autres concerts dans la même ville.

Comme on interroge deux « copies » d'une même table, il faut les renommer avec des alias. Voici par exemple comment afficher les spectateurs et leurs parrains associés qui sont dans la même ville :

```
SELECT s1.nom AS spectateur, s1.ville, s2.nom AS parrain
FROM Spectateur s1
JOIN Spectateur s2 ON s1.id_parrain = s2.id_spectateur
WHERE s1.ville = s2.ville;
```

	spectateur	ville	parrain
1	Alice	Paris	Emma
2	Chloé	Bordeaux	Jules
3	Gaëlle	Lyon	Bob
4	Inès	Lille	Farid

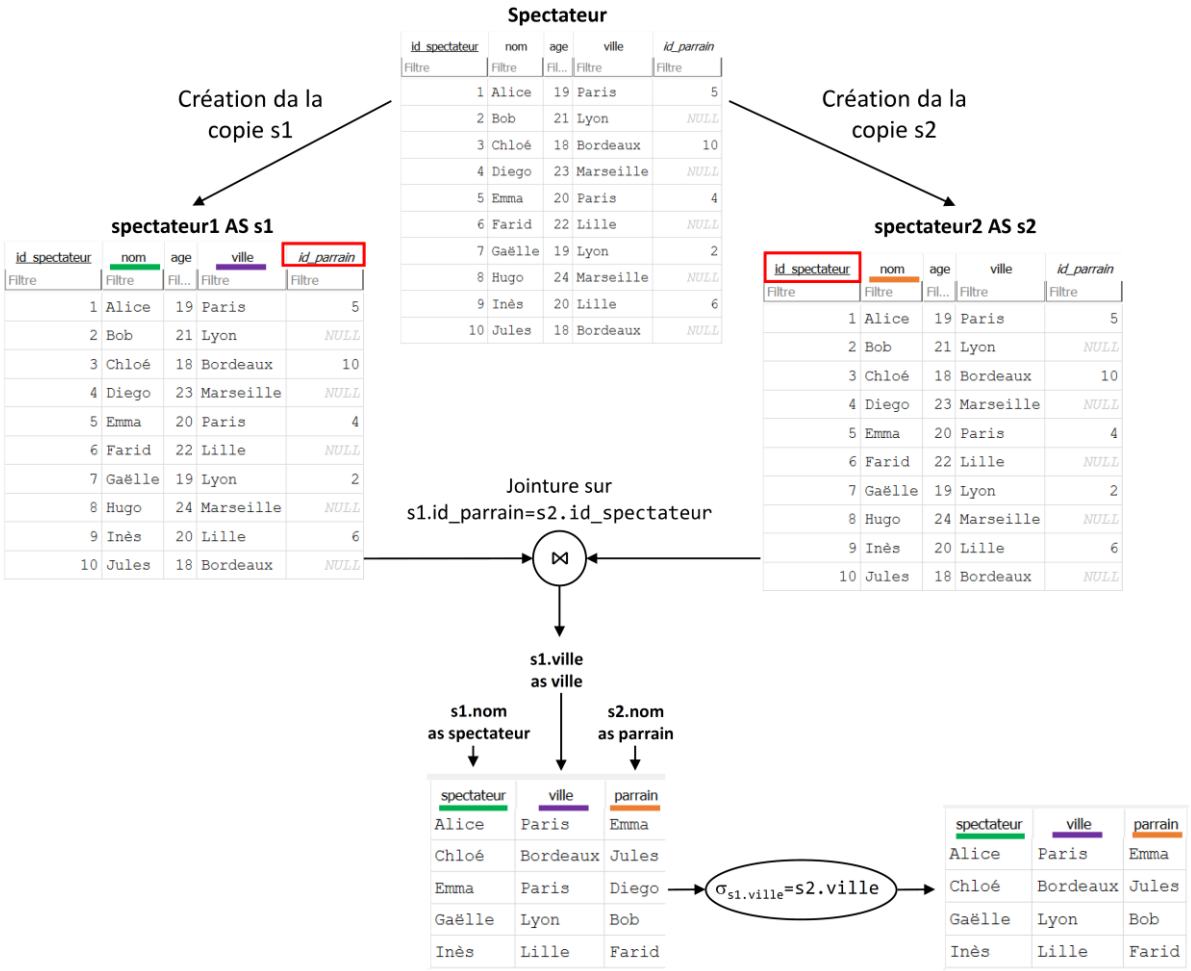


Figure 24 : Exemple d'auto-jointure

IV.5.9. Le problème des paires réflexives et symétriques dans les auto-jointures

Quand on fait une auto-jointure, on relie une table à elle-même. Mais parfois, la relation qu'on étudie est symétrique : les deux lignes qu'on relie jouent le même rôle.

Par exemple, si on cherche à lister les spectateurs d'une même ville en exécutant la commande suivante :

```
SELECT s1.nom AS spectateur1,
       s2.nom AS spectateur2,
       s1.ville
FROM Spectateur s1
JOIN Spectateur s2 ON s1.ville = s2.ville;
```

	spectateur1	spectateur2	ville
1	Alice	Alice	Paris
2	Alice	Emma	Paris
3	Bob	Bob	Lyon
4	Bob	Gaëlle	Lyon
5	Chloé	Chloé	Bordeaux
6	Chloé	Jules	Bordeaux
7	Diego	Diego	Marseille
8	Diego	Hugo	Marseille
.....	....	....	

… le SGBD va associer chaque spectateur à tous les autres dans la même ville, y compris lui-même :

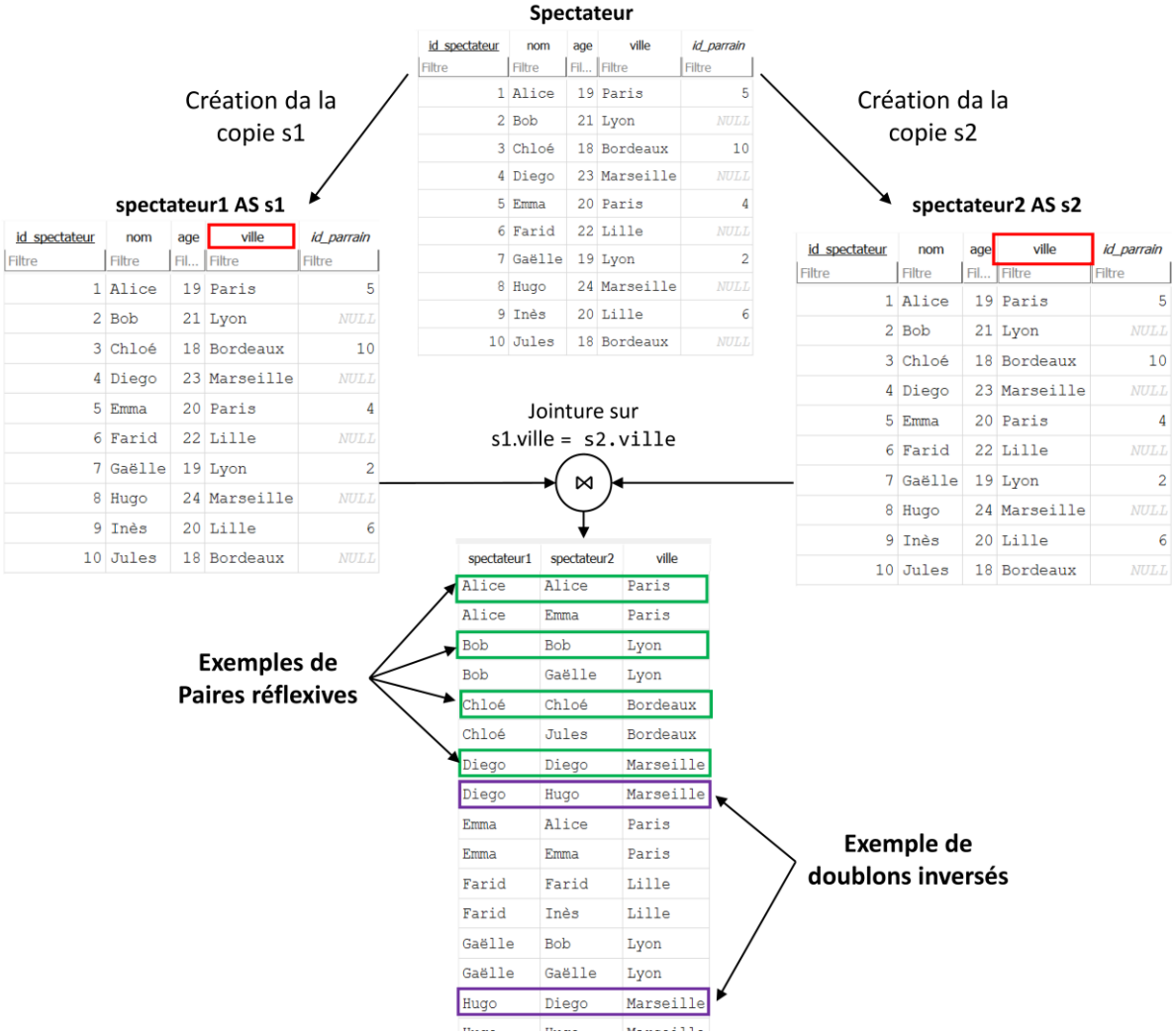


Figure 25 : Exemples de paires réflexives et de doublons inversés résultat d'une auto-jointure

Ici, on a deux problèmes :

- Les **paires réflexives** (Alice–Alice, Emma–Emma) : un spectateur lié à lui-même ;
- Les **doublons inversés** (Alice–Emma et Emma–Alice) : la même paire apparaît deux fois.

Afin d'éliminer les paires réflexives, il faut éviter les associations d'un spectateur avec lui-même. Il suffit d'ajouter une condition d'inégalité dans la jointure :

```
SELECT s1.nom AS spectateur1,
       s2.nom AS spectateur2,
       s1.ville
FROM Spectateur s1
JOIN Spectateur s2 ON s1.ville = s2.ville
AND s1.id_spectateur <> s2.id_spectateur;
```

spectateur1	spectateur2	ville
Alice	Emma	Paris
Bob	Gaëlle	Lyon
Chloé	Jules	Bordeaux
Diego	Hugo	Marseille
Emma	Alice	Paris
Farid	Inès	Lille
Gaëlle	Bob	Lyon
Hugo	Diego	Marseille
Inès	Farid	Lille
Jules	Chloé	Bordeaux
Zoé	Karim	Nantes
Karim	Zoé	Nantes

Afin d'éliminer les doublons inversés, il faut que chaque paire (par exemple la paire Alice – Emma) n'apparaisse qu'une seule fois. Pour faire cela, on impose un ordre arbitraire entre les deux lignes de la jointure, par exemple : ne garder que la paire où l'identifiant du premier est plus petit que celui du second. Pour cela, on remplace le <> par < :

```
SELECT s1.nom AS spectateur1,
       s2.nom AS spectateur2,
       s1.ville
FROM Spectateur s1
JOIN Spectateur s2 ON s1.ville = s2.ville
AND s1.id_spectateur < s2.id_spectateur;
```

spectateur1	spectateur2	ville
Alice	Emma	Paris
Bob	Gaëlle	Lyon
Chloé	Jules	Bordeaux
Diego	Hugo	Marseille
Farid	Inès	Lille
Zoé	Karim	Nantes

Il ne reste qu'une seule paire unique par combinaison, et aucune réflexive.

Pour synthétiser, sans une auto-jointure, il arrive qu'on compare chaque ligne à toutes les autres, y compris à elle-même. Pour éviter les paires réflexives (A–A) et les doublons inversés (A–B / B–A), on ajoute une condition stricte (< ou >) sur les identifiants. Cela impose un ordre et garde une seule occurrence par paire.

IV.5.10. Le rôle des index dans une jointure

Pour information (les index ne sont pas au programme), un **index** en base de données est une structure de données auxiliaire qui permet d'accéder plus rapidement aux lignes correspondant à une valeur donnée, sans avoir à lire toute la table. C'est exactement comme l'index d'un livre : pas besoin de lire toutes les pages, il suffit de rechercher directement la bonne entrée.

En SQL, une clé primaire est automatiquement indexée et une clé étrangère est souvent indexée (ou peut l'être explicitement). Ainsi, une jointure sur des clés profite naturellement de ces index pour être rapide.

Lorsqu'on écrit une jointure du type :

```
SELECT *
FROM Artiste
JOIN Participation ON Artiste.id_artiste = Participation.id_artiste;
```

…le moteur SQL cherche pour chaque `Artiste.id_artiste` de la table `Artiste` toutes les lignes correspondantes dans `Participation.id_artiste`.

Si `Participation.id_artiste` est indexée, le moteur peut accéder directement aux bonnes lignes sans lire toute la table. C'est là que les structures d'index (B-tree, B+tree, hash) interviennent.

Un **B-tree** (ou **B+tree**, une version optimisée) est un arbre de recherche équilibré. Chaque nœud contient un ensemble trié de valeurs-clés, on peut chercher une valeur en O(log n) (rapide, logarithmique) et il permet aussi de parcourir les valeurs dans l'ordre (`ORDER BY`, `BETWEEN`, etc.). C'est l'index par défaut pour les clés primaires et étrangères.

Lors d'une jointure relationnelle, le moteur lit un enregistrement dans la table 1 (par la clé primaire), puis il utilise l'index B+tree de la clé étrangère dans la table 2 pour retrouver rapidement les lignes correspondantes. Ainsi, la jointure est très rapide, même sur des tables volumineuses.

Une **table de hachage** permet d'associer chaque valeur à son adresse mémoire. Le temps d'accès est constant en moyenne : O(1). Cependant, on ne peut pas l'utiliser pour les comparaisons d'ordre (<, >, `BETWEEN`, `ORDER BY`), seulement pour des égalités exactes. Certains moteurs (comme PostgreSQL ou SQL Server) utilisent des hash pour les jointures, qui reposent sur ce principe quand la condition est une simple égalité (=).

Les jointures logiques sur des attributs quant à elles ne sont pas indexées. SQL doit comparer toutes les lignes de la table1 avec toutes celles de la table2, soit O(n×m). C'est donc beaucoup plus lent.

Pour conclure, quand on fait une jointure sur des clés, la base peut utiliser ses index, qui sont souvent des arbres B+ ou des tables de hachage, pour aller chercher les lignes correspondantes sans tout relire. Si on joint sur des colonnes non indexées, la base doit tout comparer, ce qui est beaucoup plus lent.

IV.5.11. Utilisation de `DISTINCT` dans une jointure

Quand on fait une jointure, il arrive qu'une ligne d'une table soit associée à plusieurs lignes de l'autre table. Des doublons peuvent donc apparaître dans les colonnes sélectionnées. `DISTINCT` permet alors de supprimer les doublons du résultat final, et de n'afficher chaque combinaison unique de colonnes qu'une seule fois.

Imaginons qu'on veuille afficher la liste des villes dans lesquelles les artistes français ont joué. Cette requête fait intervenir trois tables :

- `Artiste(id_artiste, nom, pays, style)`
- `Participation(id_artiste, id_concert, cachet)`
- `Concert(id_concert, date, ville, lieu)`

Avec la requête suivante :

```
SELECT a.nom, c.ville
FROM Artiste AS a
JOIN Participation AS p ON a.id_artiste = p.id_artiste
JOIN Concert AS c ON c.id_concert = p.id_concert
WHERE a.pays = 'France';
```

On obtient une table contenant des doublons sur le nom des artistes (« Jul ») ayant joué plusieurs fois dans la même ville.

nom	ville
Aya Nakamura	Paris
Aya Nakamura	Bordeaux
Daft Punk	Lyon
Orelsan	Marseille
Orelsan	Paris
Orelsan	Lyon
PNL	Marseille
Jul	Marseille
Jul	Marseille
David Guetta	Paris
David Guetta	Lyon
Jain	Bordeaux
Jain	Paris

Figure 26 : Résultat de la commande SQL

On voit en effet qu'il a joué plusieurs fois à Marseille :

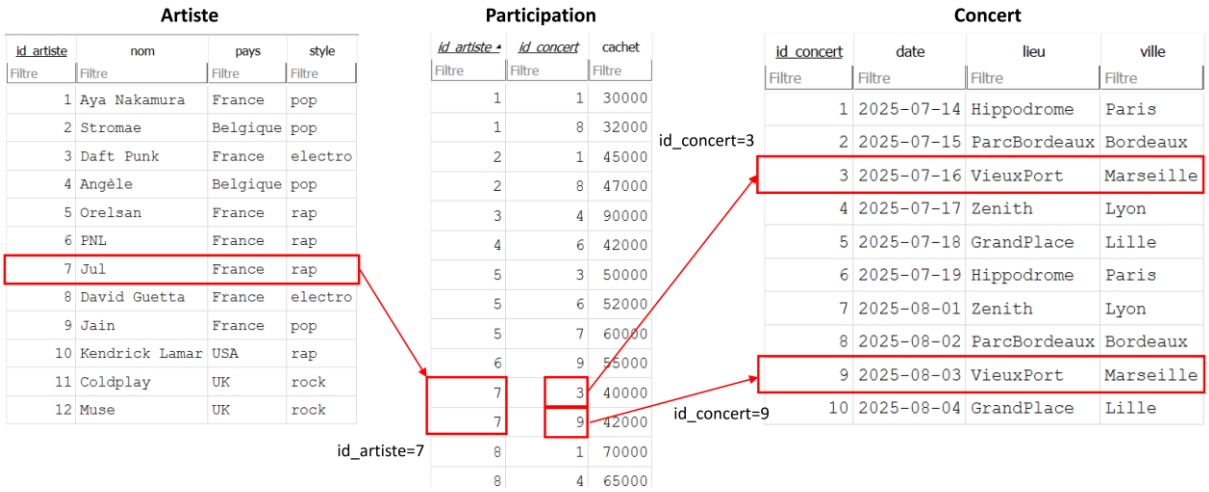


Figure 27 : L'artiste "Jul" a joué deux fois à Marseille

Pour éviter les doublons, on peut ajouter le mot clé `DISTINCT` :

```
SELECT DISTINCT a.nom, c.ville
FROM Artiste AS a
JOIN Participation AS p ON a.id_artiste = p.id_artis
JOIN Concert AS c ON c.id_concert = p.id_concert
WHERE a.pays = 'France';
```

nom	ville
Aya Nakamura	Paris
Aya Nakamura	Bordeaux
Daft Punk	Lyon
Orelsan	Marseille
Orelsan	Paris
Orelsan	Lyon
PNL	Marseille
Jul	Marseille
David Guetta	Paris
David Guetta	Lyon
Jain	Bordeaux
Jain	Paris

Figure 28 : Résultat de la commande SQL avec DISTINCT

IV.6. Les opérateurs ensemblistes

Jusqu'à présent, nous avons appris à lier plusieurs tables grâce aux jointures. Une jointure permet de fusionner les informations de plusieurs tables ligne par ligne, selon une relation logique ou une clé commune. Autrement dit, les jointures permettent de combiner

horizontalement les données : on ajoute des colonnes provenant d'autres tables pour enrichir les résultats.

Mais parfois, on souhaite combiner des ensembles complets de résultats, sans forcément les relier par une clé, comme en mathématiques quand on manipule des ensembles : on veut réunir, croiser, ou soustraire des résultats.

Les **opérateurs ensemblistes** permettent de combiner les résultats de plusieurs requêtes `SELECT` pour en faire un seul ensemble de données. Ils permettent donc une autre forme de combinaison, cette fois verticale : on empile ou on compare les ensembles de lignes issues de différentes requêtes.

Ils s'appliquent uniquement à des résultats de requêtes ayant le même nombre de colonnes et des types compatibles (par exemple deux colonnes texte, deux colonnes numériques, etc.).

IV.6.1. Union (`UNION`), intersection (`INTERSECT`) et différence (`EXCEPT`)

Les trois opérateurs principaux sont **UNION, INTERSECT et EXCEPT** :

Opérateur	Rôle	Analogie mathématique
`UNION`	Combine les lignes des deux ensembles en supprimant les doublons.	∪ (union)
`INTERSECT`	Conserve seulement les lignes communes aux deux ensembles.	∩ (intersection)
`EXCEPT`	Conserve les lignes de la 1ère requête absentes de la seconde.	\ (différence)

Tableau 5 : Principaux opérateurs ensemblistes

Ces trois opérateurs retirent les doublons par défaut. Pour conserver les doublons, on ajoute `ALL` (`UNION ALL`, `INTERSECT ALL`, `EXCEPT ALL`).

Voici quelques exemples qui utilisent les tables `Spectateur` et `Concert` :

id_spectateur	nom	age	ville	id_parrain
Filtre	Filtre	Fil...	Filtre	Filtre
1	Alice	19	Paris	5
2	Bob	21	Lyon	NULL
3	Chloé	18	Bordeaux	10
4	Diego	23	Marseille	NULL
5	Emma	20	Paris	4
6	Farid	22	Lille	NULL
7	Gaëlle	19	Lyon	2
8	Hugo	24	Marseille	NULL
9	Inès	20	Lille	6
10	Jules	18	Bordeaux	NULL
11	Zoé	25	Nantes	NULL
12	Karim	22	Nantes	NULL

id_concert	date	lieu	ville
Filtre	Filtre	Filtre	Filtre
1	2025-07-14	Hippodrome	Paris
2	2025-07-15	ParcBordeaux	Bordeaux
3	2025-07-16	VieuxPort	Marseille
4	2025-07-17	Zenith	Lyon
5	2025-07-18	GrandPlace	Lille
6	2025-07-19	Hippodrome	Paris
7	2025-08-01	Zenith	Lyon
8	2025-08-02	ParcBordeaux	Bordeaux
9	2025-08-03	VieuxPort	Marseille
10	2025-08-04	GrandPlace	Lille

Figure 29 : Tables Spectateur (gauche) et Concert (droite)

Exemple	Commande SQL	Résultat
Liste des villes des spectateurs et celles où ont lieu les concerts. Les valeurs renvoyées sont uniques.	`SELECT ville FROM Spectateur` `UNION` `SELECT ville FROM Concert` `ORDER BY ville;`	ville Bordeaux Lille Lyon Marseille Nantes Paris
Liste les villes présentes dans les deux tables.	`SELECT ville FROM Spectateur` `INTERSECT` `SELECT ville FROM Concert` `ORDER BY ville;`	ville Bordeaux Lille Lyon Marseille Paris
Liste les villes des spectateurs qui ne reçoivent pas de concert.	`SELECT Spectateur.ville FROM Spectateur` `EXCEPT SELECT Concert.ville FROM Concert;`	ville Nantes

Tableau 6 : Exemples de commandes SQL avec les opérateurs `UNION`, `INTERSECT` ET `EXCEPT`

`EXCEPT` retourne un **ensemble distinct de valeurs** (ici, juste « Nantes », même si plusieurs spectateurs y habitent). Ces résultats font **forcément partie de l'attribut étudié**.

IV.6.2. `IN` et `NOT IN`

Les opérations réalisées avec les opérateurs `UNION` et `EXCEPT` peuvent se faire avec les opérateurs `IN` et `NOT IN`. Cependant, ces deux opérateurs **ne renvoient pas de résultats distincts**.

Exemple	Commande SQL	Résultat
Liste des villes des spectateurs et celles où ont lieu les concerts. Les valeurs renvoyées ne sont pas uniques.	`SELECT ville FROM Spectateur` `WHERE ville IN (SELECT ville FROM Concert);` `(équivalent à l'opérateur UNION)`	ville Paris Lyon Bordeaux Marseille Paris Lille Lyon Marseille Lille Bordeaux
Liste les villes des spectateurs qui ne reçoivent pas de concert. Les valeurs retournées ne sont pas uniques.	`SELECT ville FROM Spectateur` `WHERE ville NOT IN (SELECT ville FROM Concert);` `(équivalent à l'opérateur EXCEPT)`	ville Nantes Nantes

Tableau 7 : Exemples de commandes SQL avec les opérateurs IN ET NOT IN

Une autre différence entre l'utilisation des opérateurs `UNION/EXCEPT` et `IN/NOT IN` est qu'avec ces derniers, **on peut sélectionner un autre attribut que celui qui est testé**.

Supposons par exemple qu'on veuille les noms des spectateurs dont la ville n'accueille aucun concert. Ici, on ne veut pas juste la ville, mais les personnes (et on accepte les répétitions s'il y en a).

```
SELECT nom, ville
FROM Spectateur
WHERE ville NOT IN (SELECT ville FROM Concert)
ORDER BY ville, nom;
```

→

nom	ville
Karim	Nantes
Zoé	Nantes

Dans ce dernier exemple, on peut sélectionner un autre attribut que celui « testé » (ici on teste ville, mais on affiche nom, ville).

IV.6.3. NOT EXISTS

Un problème peut survenir avec l'utilisation de `NOT IN` lorsque la sous-requête renvoie au moins un `NULL`. En effet, dans ce cas aucune comparaison n'est vraie et le résultat est vide parce qu'en SQL, `NULL` signifie valeur inconnue.

Prenons l'exemple de la table `Spectateur`, dont certains spectateurs ont un parrain et d'autres n'en n'ont pas (`id_parrain=NULL`).

→

id_spectateur	nom	age	ville	id_parrain
1	Alice	19	Paris	5
2	Bob	21	Lyon	NULL
3	Chloé	18	Bordeaux	10
4	Diego	23	Marseille	NULL
5	Emma	20	Paris	4
6	Farid	22	Lille	NULL
7	Gaëlle	19	Lyon	2
8	Hugo	24	Marseille	NULL
9	Inès	20	Lille	6
10	Jules	18	Bordeaux	NULL
11	Zoé	25	Nantes	NULL
12	Karim	22	Nantes	NULL

Figure 30 : Table Spectateur

Essayons de lister les spectateurs qui ne sont le parrain de personn spectateurs dont l'`id_spectateur` n'apparaît jamais dans la colonne `id_parrain` :

```
SELECT nom
FROM Spectateur
WHERE id_spectateur NOT IN (SELECT id_parrain FROM Spectateur);
```

… cette commande ne retourne aucun résultat !

En effet, la sous-requête `SELECT id_parrain FROM Spectateur` renvoie une table contenant à la fois des valeurs connues (5,10,4,2,6) et des valeurs `NULL`. La condition `id_spectateur NOT IN (5,NULL,10,NULL,4,…)` revient à tester, pour chaque valeur de `id_spectateur`, la négation d'un ensemble de comparaisons :

```
NOT (id_spectateur=5 OR id_spectateur=NULL OR id_spectateur=10 OR …)
```

Lorsque par exemple `id_spectateur=1`, cela revient à faire les tests :

```
NOT (1=5 OR 1=NULL OR 1=10 OR …)
```

La comparaison `1=5` renvoie `FALSE`, mais la comparaison `1=NULL` renvoie `UNKNOWN`. Ainsi, le test devient :

```
NOT (FALSE OR UNKNOWN OR FALSE OR …) = NOT(FALSE) OR NOT(UNKNOWN) OR NOT(FALSE) …
```

Et le problème survient sur la comparaison `NOT(UNKNOWN)` … car `UNKNOWN` peut à la fois être `TRUE` ou `FALSE`.

Le résultat est donc ni vrai ni faux, donc la ligne est éliminée :

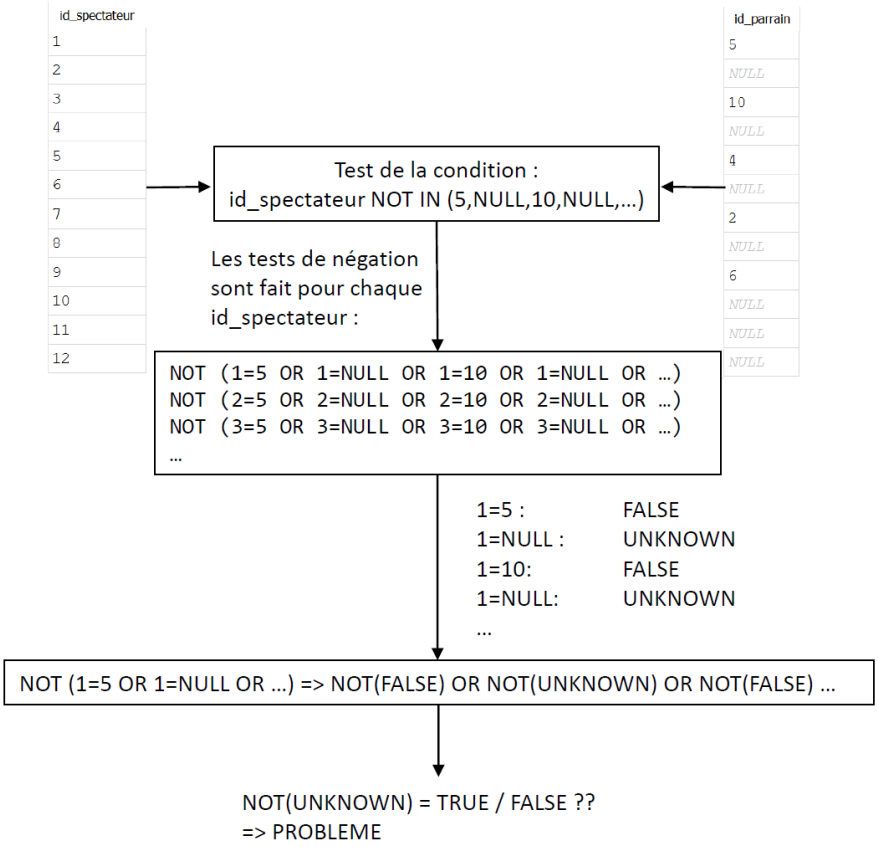


Figure 31 : Utilisation de NOT IN avec des valeurs NULL

La conséquence est que tant qu'un `NULL` figure dans la liste, toutes les comparaisons deviennent indéterminées. Le résultat global est qu'aucune ligne n'est renvoyée.

La solution est d'utiliser `NOT EXISTS` :

```
SELECT s.nom
FROM Spectateur s
WHERE NOT EXISTS (
  SELECT 1
  FROM Spectateur p WHERE p.id_parrain = s.id_spectateur
);
```

→

nom
Alice
Chloé
Gaëlle
Hugo
Inès
Zoé
Karim

Le fonctionnement est le suivant :

- Pour une ligne donnée de s (par exemple `s.nom = Alice`), SQL exécute la requête contenue dans `NOT EXISTS ()` en remplaçant `s.id_spectateur` par la valeur qui lui correspond pour le nom sélectionné (1 pour Alice) :

```
SELECT 1
FROM Spectateur p
WHERE p.id_parrain = 1;
```

- Cette requête retourne 1 si elle est vraie, c'est-à-dire s'il y a un `p.id_parrain=1` dans la liste des `id_parrain` {5, NULL, 10, NULL, 4, …} ou sinon elle retourne un résultat vide.

- Si le résultat est 1 (résultat non vide), `NOT EXISTS()` retourne FALSE, sinon il retourne TRUE.

- A chaque fois que `NOT EXISTS()` est `TRUE`, le nom est donc sélectionné et il n'est pas sélectionné dans le cas contraire.

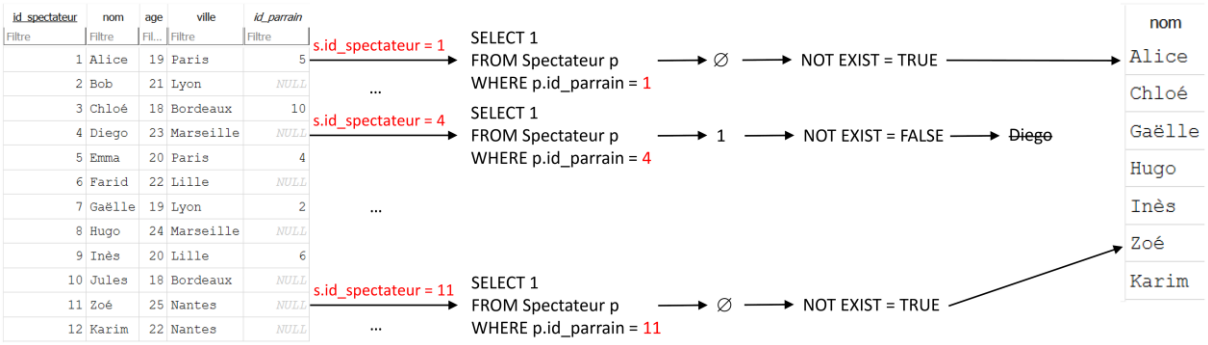


Figure 32 : Utilisation de la requête NOT EXISTS

IV.7. Les fonctions d'agrégation

Le langage SQL propose les **fonctions d'agrégation**, qui permettent de calculer des statistiques telles que le nombre, la moyenne, la somme, le minimum ou le maximum de certaines valeurs.

Fonction	Exemple	Résultat
COUNT()	Nombre total de concerts : `SELECT COUNT(*) AS nb_concerts FROM Concert;`	nb_concerts 10
	Nombre moyen de concerts par artiste : `SELECT COUNT(DISTINCT id_concert)*1.0 / COUNT(DISTINCT id_artiste) AS moyenne FROM Participation;`	moyenne 0.833333333333333
	Compter combien de numéros de sécu sont enregistrés : `SELECT COUNT(numero_secu) FROM Identite;`	COUNT(numero_secu) 5
MIN()	Cachet le plus faible parmi tous les artistes : `SELECT MIN(cachet) AS cachet_min FROM Participation;`	cachet_min 25000
MAX()	Cachet le plus élevé versé à un artiste : `SELECT MAX(cachet) AS cachet_max FROM Participation;`	cachet_max 120000
SUM()	Montant total des cachets versés (tous concerts confondus) : `SELECT SUM(cachet) AS total_cachets FROM Participation;`	total_cachets 1304000
AVG()	Cachet moyen versé aux artistes : `SELECT AVG(cachet) AS cachet_moyen FROM Participation;`	cachet_moyen 59272.7272727273
	Moyenne d'âge des spectateurs : `SELECT AVG(age) AS age_moyen FROM Spectateur;`	age_moyen 20.9166666666667
Divers	Écart entre cachet max et min `SELECT MAX(cachet) - MIN(cachet) AS ecart_cachets FROM Participation;`	ecart_cachets 95000

Il est possible de combiner plusieurs agrégats dans un même requête :

```
SELECT
  COUNT(*) AS nb_concerts,
  MIN(date) AS premier_concert,
  MAX(date) AS dernier_concert
FROM Concert;
```

→

nb_concerts	premier_concert	dernier_concert
10	2025-07-14	2025-08-04

```
SELECT
  COUNT(*) AS nb_identites,
  COUNT(numero_secu) AS nb_numero_secu
FROM Identite;
```

→

nb_identites	nb_numero_secu
5	5

IV.8. Le groupement des données : `GROUP BY` et `HAVING`

Jusqu'à présent, les fonctions d'agrégation (`COUNT`, `AVG`, `SUM`, `MIN`, `MAX`) s'appliquaient à toute la table. Mais souvent, on souhaite résumer les données par catégorie : nombre de concerts par ville, cachet moyen par artiste, nombre de spectateurs par tranche d'âge, etc.

C'est exactement le rôle de GROUP BY : il permet de créer des sous-groupes **avant d'appliquer les fonctions d'agrégation** (notées f ci-dessous) :

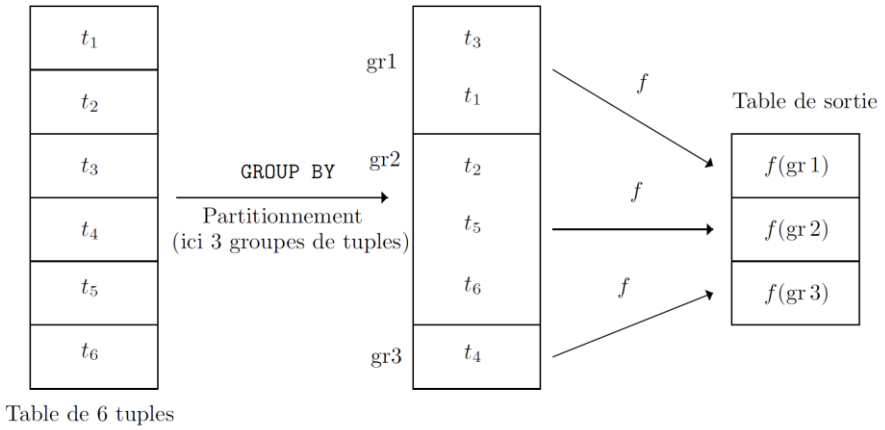


Figure 33 : Effet de l'instruction GROUP BY

IV.8.1. Principe de base

La syntaxe générale est la suivante :

```
SELECT colonne_groupe, fonction_agrégation(colonne)
FROM table
GROUP BY colonne_groupe;
```

Par exemple, pour afficher le nombre de concerts par ville, on commence par regrouper les concerts par ville, puis on compte combien il y a de concerts dans chaque groupe :

```
SELECT ville, COUNT(*) AS nb_concerts
FROM Concert
GROUP BY ville;
```

ville	nb_concerts
Bordeaux	2
Lille	2
Lyon	2
Marseille	2
Paris	2

Les deux commandes ci-dessous renvoient la même chose :

```
SELECT ville
FROM Concert
GROUP BY ville;
```

```
SELECT DISTINCT ville
FROM Concert;
```

ville
Paris
Bordeaux
Marseille
Lyon
Lille

… sauf qu'avec `GROUP BY`, des choses restent cachées et il est possible d'aller chercher des informations sur les attributs qui lui sont rattachés (comme le nombre de concerts dans chaque ville).
Quand on utilise `GROUP BY`, seules deux sortes de colonnes peuvent apparaître dans le `SELECT` : celles qui figurent dans le `GROUP BY` et celles qui sont calculées par une fonction d'agrégation. Sinon, SQL ne saurait pas quelle valeur choisir pour une colonne non groupée.

Voici quelques exemples :

Exemple	Commande SQL	Résultat
Somme des cachets par ville.	`SELECT c.ville, SUM(p.cachet) AS total_cachet` `FROM Participation p` `JOIN Concert c ON p.id_concert = c.id_concert` `GROUP BY c.ville;`	ville | total_cachet Bordeaux | 184000 Lille | 333000 Lyon | 335000 Marseille | 187000 Paris | 265000
Cachet moyen par artiste.	`SELECT a.nom, AVG(p.cachet) AS cachet_moyen` `FROM Artiste a` `JOIN Participation p ON a.id_artiste = p.id_artiste` `GROUP BY a.nom;`	nom | cachet_moyen Angèle | 42000.0 Aya Nakamura | 31000.0 Coldplay | 80000.0 Daft Punk | 90000.0 David Guetta | 67500.0 Jain | 25500.0 Jul | 41000.0 Kendrick Lamar | 120000.0 Muse | 86500.0 … …
Nombre de billets vendus par concert.	`SELECT id_concert, COUNT(*) AS nb_billets` `FROM Billet` `GROUP BY id_concert;`	id_concert | nb_billets 1 | 2 2 | 2 3 | 2 4 | 2 5 | 1 … …
Nombre d'artistes avec un numéro de sécurité sociale.	`SELECT COUNT(*) AS nb_artistes_avec_secu` `FROM Artiste a` `JOIN Identite i ON a.id_artiste = i.id_artiste;`	nb_artistes_avec_secu 5

IV.8.2. Combiner GROUP BY avec WHERE : Filtrer avant le regroupement

On peut filtrer **avant le regroupement** et l'application des fonctions d'agrégation grâce à WHERE. Les fonctions d'agrégation s'appliquent donc ici sur des lignes :

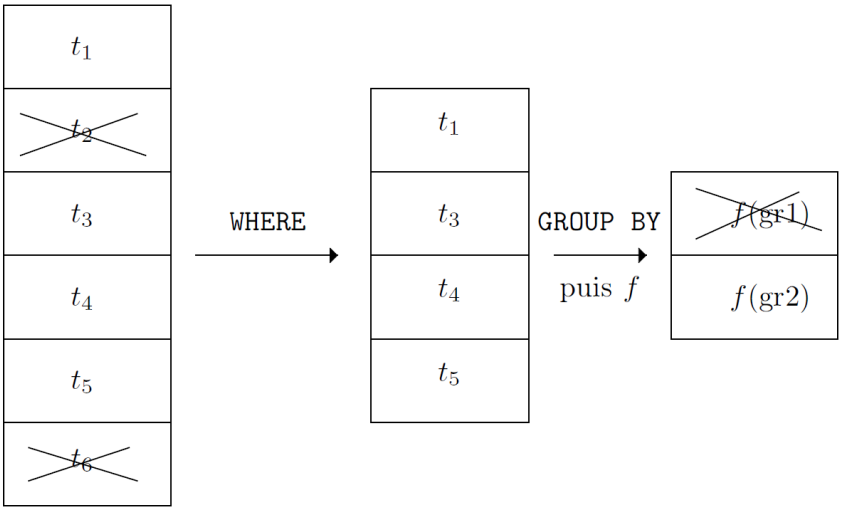


Figure 34 : Combiner GROUP BY avec WHERE

Par exemple, si on souhaite obtenir la somme des cachets uniquement à Paris :

```
SELECT c.ville, SUM(p.cachet) AS total_cachet
FROM Participation p
JOIN Concert c ON p.id_concert = c.id_concert
WHERE c.ville = 'Paris'
GROUP BY c.ville;
```

ville	total_cachet
Paris	265000

Ici, `WHERE` sélectionne d'abord les concerts à Paris, puis `GROUP BY` fait le calcul par ville (ici, une seule ville) :

Figure 35 : Combiner GROUP BY avec WHERE

IV.8.3. Combiner `GROUP BY` avec `HAVING` : filtrer après le regroupement

On peut filtrer **après le regroupement** grâce à `HAVING`. `HAVING` est le pendant de `WHERE`, mais il s'applique après l'agrégation, donc sur les groupes et non les lignes :

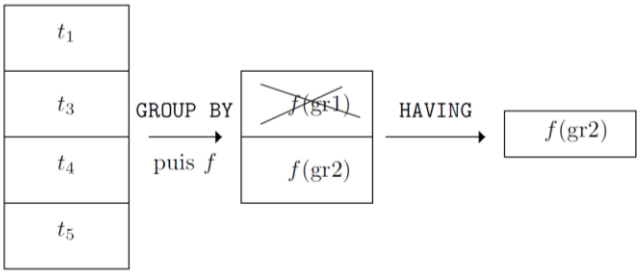


Figure 36 : Combiner GROUP BY avec HAVING

La syntaxe générale est la suivante :

```
SELECT colonne_groupe, fonction_agrégation(colonne)
FROM table
GROUP BY colonne_groupe
HAVING condition_sur_agrégat;
```

Par exemple, pour obtenir les villes ayant plus de 1 concert :

```
SELECT ville, COUNT(*) AS nb_concerts
FROM Concert
GROUP BY ville
HAVING COUNT(*) > 1;
```

→

ville	nb_concerts
Bordeaux	2
Lille	2
Lyon	2
Marseille	2
Paris	2

Ici, `HAVING` filtre sur le résultat de la fonction d'agrégation `COUNT(*)`.

Voici quelques exemples :

Exemple	Commande SQL	Résultat
Artistes dont le cachet moyen est > 40 000EUR	`SELECT a.nom, AVG(p.cachet) AS cachet_moyen` `FROM Artiste a` `JOIN Participation p` `ON a.id_artiste = p.id_artiste` `GROUP BY a.nom` `HAVING AVG(p.cachet) > 40000;` *Ici, `HAVING` filtre sur `AVG(p.cachet)`*	nom / cachet_moyen Angèle 42000.0 Coldplay 80000.0 Daft Punk 90000.0 David Guetta 67500.0 Jul 41000.0 Kendrick Lamar 120000.0 Muse 86500.0 Orelsan 54000.0 PNL 55000.0 Stromae 46000.0
Moyenne d'âge par ville, uniquement si moyenne > 20 ans.	`SELECT ville, AVG(age) AS age_moyen` `FROM Spectateur` `GROUP BY ville` `HAVING AVG(age) > 20;` *Ici, `HAVING` filtre sur `AVG(age)`*	ville / age_moyen Lille 21.0 Marseille 23.5 Nantes 23.5

IV.8.4. Comparaison des propriétés de `WHERE` et `HAVING`

Les instructions `WHERE` et `HAVING` réalisent exactement les mêmes opérations (élimination de lignes d'une table selon un critère), mais l'une (`WHERE`) intervient sur les lignes avant le calcul de la fonction d'agrégation tandis que l'autre (`HAVING`) intervient après sur les groupes :

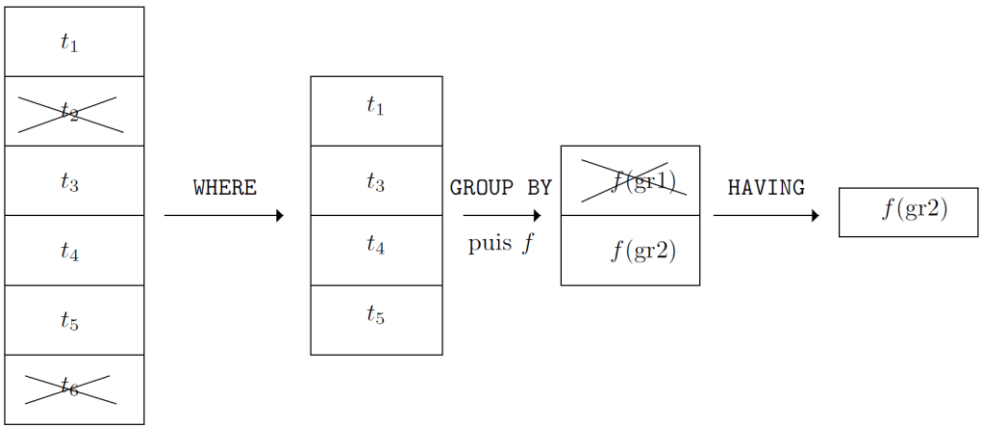


Figure 37 : Combiner WHERE, GROUP BY et HAVING

Voici un résumé des propriétés de `WHERE` et `HAVING` lorsqu'ils sont utilisés avec `GROUP BY` :

Aspect	WHERE	HAVING
Filtrage avant ou après agrégation ?	Avant regroupement	Après regroupement
Sur quoi agit-il ?	Sur les **lignes**	Sur les **groupes**
Peut-il contenir une fonction d'agrégation ?	Non	Oui
Exemple	`WHERE ville = 'Paris'`	`HAVING COUNT(*) > 1`

IV.8.5. Combinaison complète : `WHERE` et `HAVING` avec `GROUP BY`

Il n'est pas toujours évident de savoir quand utiliser les clauses `WHERE`, `GROUP BY` et `HAVING`, et dans quel ordre les utiliser. Ces clauses jouent chacune un rôle très différent, et la clé est de comprendre à quel moment du traitement SQL ils interviennent.

La requête SQL est toujours écrite dans cet ordre :

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
```

… mais SQL ne l'exécute pas dans cet ordre.

Voici l'ordre logique réel :

Étape	Clause	Rôle
1	`FROM / JOIN`	On forme la table de travail (avec toutes les lignes nécessaires)
2	`WHERE`	On filtre les **lignes individuelles** (avant regroupement)
3	`GROUP BY`	On regroupe les lignes par valeurs communes
4	`HAVING`	On filtre les **groupes** créés à l'étape précédente
5	`SELECT`	On calcule les résultats finaux (agrégats, colonnes, etc.)
6	`ORDER BY`	On trie les résultats

Prenons un exemple : Afficher le total des cachets par ville, par ordre décroissant, mais seulement pour les concerts à partir du 20 juillet 2025, et ne garder que les villes où la somme dépasse 80 000 EUR.

On souhaite obtenir un résultat de la forme :

ville	total_cachet
ville 1	total 1
ville 2	total 2
…	…

Il nous faut donc :

- les cachets des artistes (table `Participation`),
- les villes et les dates où se déroulent les concerts (table `Concert`).

On travaille donc avec les tables `Participation` et `Concert`.

Étape 1 – FROM / JOIN :

On forme la table de travail en réalisant des jointures relationnelles. Comme on a besoin des cachets et des villes on va relier la table `Concert` à la table `Participation` à l'aide de la clé `id_concert`.

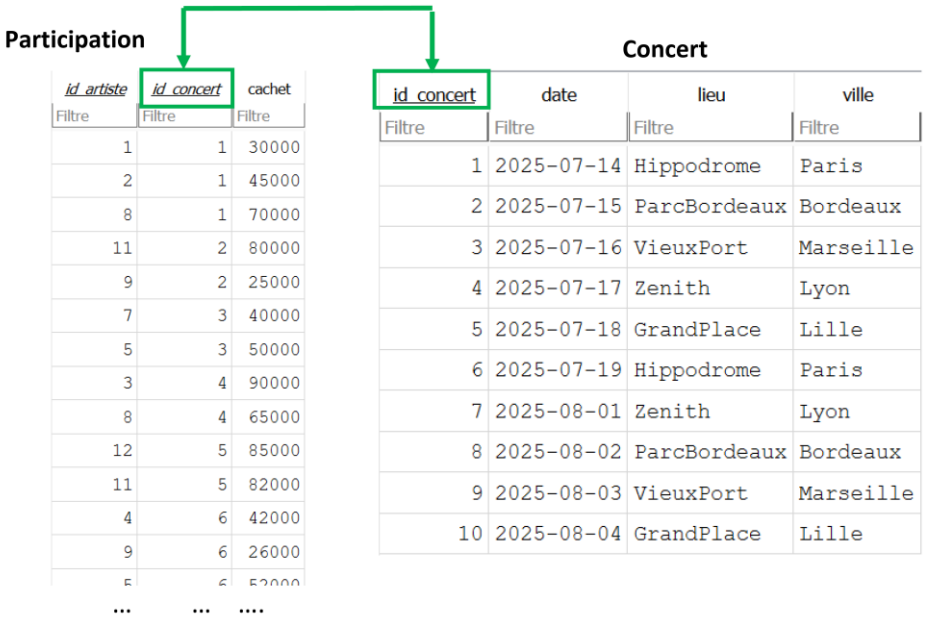


Figure 38 : Tables et clés utilisées pour les jointures relationnelles

La commande SQL utilisée sera donc :

```
FROM Participation p
JOIN Concert AS c ON p.id_concert = c.id_concert
```

La table résultant de cette jointure ressemble donc à cela :

id_artiste	id_concert	cachet	id_concert	date	lieu	ville
1	1	30000	1	2025-07-14	Hippodrome	Paris
2	1	45000	1	2025-07-14	Hippodrome	Paris
8	1	70000	1	2025-07-14	Hippodrome	Paris
11	2	80000	2	2025-07-15	ParcBordeaux	Bordeaux
...	...	...	...	...	...	...

Figure 39 : Table résultat de la jointure

Étape 2 – WHERE :

On filtre les données de la table obtenue. Dans notre exemple, il y a deux filtres à appliquer :

- ne garder que les dates après le 20 juillet 2025,
- ne garder que les sommes dont le total des cachets est > 80 000 EUR.

Mais à ce stade, on ne peut pas utiliser `WHERE` pour filtrer sur les sommes car elles n'existent pas encore. En effet, quand WHERE est exécuté le `SUM()` n'a pas été encore calculé car il est calculée par une fonction d'agrégation qui est appliquée uniquement après `GROUP BY`.

Ici, on va donc utiliser `WHERE` pour filtrer les dates afin de ne garder que les informations qui sont après le 20 juillet 2025 :

```
WHERE c.date >= '2025-07-20'
```

La table à ce stade ressemble à cela :

id_artiste	id_concert	cachet	id_concert	date	lieu	ville
10	7	120000	7	2025-08-01	Zenith	Lyon
5	7	60000	7	2025-08-01	Zenith	Lyon
2	8	47000	8	2025-08-02	ParcBordeaux	Bordeaux
1	8	32000	8	2025-08-02	ParcBordeaux	Bordeaux
6	9	55000	9	2025-08-03	VieuxPort	Marseille
7	9	42000	9	2025-08-03	VieuxPort	Marseille
11	10	78000	10	2025-08-04	GrandPlace	Lille
12	10	88000	10	2025-08-04	GrandPlace	Lille

Figure 40 : Table après filtrage des dates avec WHERE

Étape 3 – `GROUP BY` :

Comme on a besoin de calculer les sommes pour chaque ville, on regroupe les concerts par ville :

```
GROUP BY c.ville
```

La table à ce stade ressemble à cela :

id_artiste	id_concert	cachet	id_concert	date	lieu	ville
2	8	47000	8	2025-08-02	ParcBordeaux	Bordeaux
11	10	78000	10	2025-08-04	GrandPlace	Lille
10	7	120000	7	2025-08-01	Zenith	Lyon
6	9	55000	9	2025-08-03	VieuxPort	Marseille

Figure 41 : Table résultat de GROUP BY c.ville

Attention : dans la table, l'attribut `cachet` ne représente pas la somme des cachets pour chaque ville mais le cachet de la première ligne contenue dans le regroupement des villes. Derrière l'attribut `cachet`, il y a en réalité les autres cachets qui sont « cachés » et qui permettront de calculer la somme des cachets par regroupement des villes. Il en va de même pour les autres attributs, en particulier pour l'attribut `id_artiste`. Ici l'id affiché est celui qui est dans la première ligne du regroupement des villes :

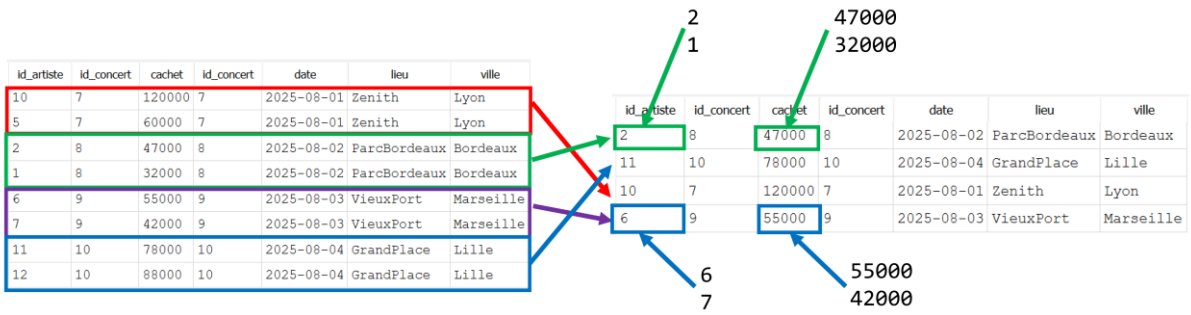


Figure 42 : Effet de l'opération GROUP BY c.ville

Étape 4 – HAVING :

On vient filtrer les groupes des villes par rapport à la fonction d'agrégation permettant d'avoir la somme des cachets, c'est-à-dire `SUM(p.cachet)` :

```
HAVING SUM(p.cachet) > 80000
```

La table à ce stade ressemble à cela : elle ne contient que les villes dont la somme des cachets est supérieure à 80 000 EUR :

id_artiste	id_concert	cachet	id_concert	date	lieu	ville
11	10	78000	10	2025-08-04	GrandPlace	Lille
10	7	120000	7	2025-08-01	Zenith	Lyon
6	9	55000	9	2025-08-03	VieuxPort	Marseille

Figure 43 : Table résultat de HAVING p.cachet > 80000

On remarque que la ville « Bordeaux » a bien été supprimée car la somme était inférieure à 80000 EUR (47000 + 32000 = 79000).

Étape 5 – SELECT :

On vient sélectionner les attributs à afficher : `ville` et `total_cachet`.

```
SELECT c.ville, SUM(p.cachet) AS total_cachet
```

→

ville	total_cachet
Lille	166000
Lyon	180000
Marseille	97000

Étape 6 – ORDER BY :

On trie les résultats :

```
ORDER BY total_cachet DESC;
```

→

ville	total_cachet
Lyon	180000
Lille	166000
Marseille	97000

La requête complète est donc :

```
SELECT c.ville, SUM(p.cachet) AS total_cachet
FROM Participation p
JOIN Concert AS c ON p.id_concert =
c.id_concert
WHERE c.date >= '2025-07-20'
GROUP BY c.ville
HAVING SUM(p.cachet) > 80000;
```

→

ville	total_cachet
Lille	166000
Lyon	180000
Marseille	97000

En conclusion :

- Utiliser `WHERE` pour filtrer des données "brutes" avant le calcul des agrégats. (`WHERE` = ligne par ligne)
- Utiliser `GROUP BY` pour regrouper les lignes par catégorie.
- Utiliser `HAVING` pour filtrer les résultats agrégés après regroupement. (`HAVING` = groupe par groupe)

IV.8.6. Utiliser des sous-requêtes dans la clause HAVING

`HAVING` ne filtre pas seulement sur des agrégats simples, mais il peut aussi comparer le résultat d'un groupe à une valeur calculée dynamiquement par une autre requête SQL.

Si par exemple on souhaite savoir quels artistes ont un cachet moyen supérieur à la moyenne générale de tous les artistes :

- On calcule le cachet moyen par artiste (`AVG(p.cachet)` avec `GROUP BY a.nom`) ;
- Puis on ne garde que ceux dont la moyenne est supérieure à la moyenne globale, qui sera calculée via une sous-requête SELECT.

```
SELECT a.nom, AVG(p.cachet) AS cachet_moyen
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
GROUP BY a.nom
HAVING AVG(p.cachet) >
  (SELECT AVG(cachet) FROM Participation);
```

→

nom	cachet_moyen
Coldplay	80000.0
Daft Punk	90000.0
David Guetta	67500.0
Kendrick Lamar	120000.0
Muse	86500.0

La sous-requête `SELECT AVG(cachet) FROM Participation` renvoie une seule valeur : la moyenne générale (47 500). Le `HAVING` compare chaque moyenne d'artiste à cette valeur globale. Seuls les artistes au-dessus de cette moyenne sont conservés.

`HAVING` ne sert pas seulement à filtrer les groupes sur des constantes. Il peut comparer chaque groupe à une valeur calculée à la volée par une autre requête SQL, ce qui rend les analyses bien plus dynamiques.

IV.8.7. Utiliser GROUP BY sur plusieurs attributs

Quand `GROUP BY` doit gérer plusieurs colonnes, le SQL crée un groupe pour chaque combinaison unique de valeurs de ces colonnes. Autrement dit, le regroupement se fait hiérarchiquement :

- D'abord par la première colonne,
- Puis, à l'intérieur, par la seconde,
- Puis la troisième, etc

Imaginons qu'on veuille savoir combien de concerts chaque artiste a donnés dans chaque ville. Les informations nécessaires se trouvent dans les tables `Artiste`, `Participation`, et `Concert`.

La requête est la suivante :

```
SELECT a.nom AS artiste, c.ville, COUNT(*)
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON p.id_concert = c.id_concert
GROUP BY a.nom, c.ville
ORDER BY a.nom, c.ville;
```

SQL crée un groupe unique pour chaque couple (artiste, ville). Il compte ensuite combien de participations existent dans chaque groupe.

artiste	ville	COUNT(*)
Angèle	Paris	1
Aya Nakamura	Bordeaux	1
Aya Nakamura	Paris	1
Coldplay	Bordeaux	1
Coldplay	Lille	2
Daft Punk	Lyon	1
David Guetta	Lyon	1
David Guetta	Paris	1
Jain	Bordeaux	1
Jain	Paris	1
Jul	Marseille	2
Kendrick Lamar	Lyon	1
Muse	Lille	2
Orelsan	Lyon	1
Orelsan	Marseille	1
Orelsan	Paris	1
PNL	Marseille	1
Stromae	Bordeaux	1
Stromae	Paris	1

Autre exemple : quel est le cachet moyen par artiste et par ville ?

```
SELECT a.nom, c.ville, AVG(p.cachet)
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON p.id_concert = c.id_concert
GROUP BY a.nom, c.ville
ORDER BY a.nom;
```

nom	ville	AVG(p.cachet)
Angèle	Paris	42000.0
Aya Nakamura	Bordeaux	32000.0
Aya Nakamura	Paris	30000.0
Coldplay	Bordeaux	80000.0
Coldplay	Lille	80000.0
Daft Punk	Lyon	90000.0
David Guetta	Lyon	65000.0
David Guetta	Paris	70000.0
Jain	Bordeaux	25000.0
...	...	...

Chaque couple (artiste, ville) forme un groupe indépendant. SQL calcule un agrégat (`AVG`) pour chacun.

Autre exemple : Liste des artistes qui ont joué dans les villes au moins deux fois.

```
SELECT a.nom, c.ville, COUNT(*) AS nb_concerts
FROM Artiste a
JOIN Participation p ON a.id_artiste = p.id_artiste
JOIN Concert c ON p.id_concert = c.id_concert
GROUP BY a.nom, c.ville
HAVING COUNT(*) >= 2
ORDER BY nb_concerts DESC;
```

artiste	ville	nb_concerts
Coldplay	Lille	2
Jul	Marseille	2
Muse	Lille	2

V) LE MODÈLE CONCEPTUEL DE DONNÉES (MCD)

Avant de créer une base de données dans un langage comme SQL, il est essentiel de réfléchir à la structure logique des informations que l'on veut stocker.

Le **modèle conceptuel des données** (ou modèle entité–association EA) permet de représenter de manière simple et visuelle les objets du monde réel (les **entités**) et les liens qui les unissent (les **associations**).

Ce modèle a été inventé par l'informaticien Peter Chen, américain d'origine taïwanaise. C'est ce modèle qui a inspiré Merise, UML, et quasiment toutes les méthodes de conception de bases de données d'aujourd'hui.

Dans ce modèle, chaque entité correspond à un objet concret (comme un artiste, un concert ou un spectateur), et chaque association exprime un lien entre ces objets (par exemple un artiste participe à un concert).

Ce modèle constitue la première étape de la conception d'une base de données : il aide à comprendre les relations entre les données avant leur mise en œuvre technique.

V.1. Entités de la base de données

Une entité représente un objet ou une notion que l'on veut stocker dans la base. Dans le cas du festival, les entités principales sont :

Entité	Description	Attributs
Artiste	Personne ou groupe se produisant au festival	id_artiste, nom, style
Concert	Événement organisé dans une ville et à une date	id_concert, date, ville
Spectateur	Personne assistant à un concert	id_spectateur, nom, ville, âge
Identité	Données administratives d'un artiste	id_identité, numéro_sécu
Participation	Relation entre Artiste et Concert	cachet
Parrainage	Relation entre spectateurs	auto-référence (id_parrain)

Figure 44 : Entités de la base de données "festival"

V.2. Principe des associations, cardinalités et schéma conceptuel

Une **association** relie deux (ou plusieurs) entités par un verbe (ou un nom de relation) qui exprime le lien et indique combien d'occurrences d'une entité peuvent être liées à une autre avec **les cardinalités conceptuelles**.

Les cardinalités se lisent de chaque côté du verbe et son notées de la manière suivante :

Notation	Lecture
(1,1)	Exactement un
(0,1)	Zéro ou un (au plus un)
(1,n)	Au moins un
(0,n)	Zéro, un ou plusieurs

Tableau 8 : Cardinalités conceptuelles

V.3. Les associations dans notre base de données

Avant d'étudier comment sont représentées les associations et les cardinalités qui leurs sont attachées, faisons une synthèse sur les différentes associations que l'on trouve dans notre base de données.

La base de données contient 6 tables liées entre elles :

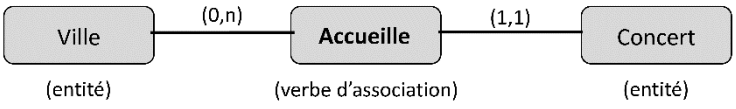
- `Artiste` : elle contient les informations sur les artistes participant au festival.
- `Identité` : elle contient l'identifiant d'identité unique de l'artiste, et éventuellement son numéro de sécurité sociale. Chaque artiste possède exactement une identité et une identité est associée à exactement un artiste ;
- `Concert` : elle contient les informations sur chaque concert organisé. Un concert se déroule dans une seule ville (pas de tournée), mais une ville peut accueillir plusieurs concerts.
- `Participation` : elle indique quels artistes jouent dans quels concerts et leur cachet. Un artiste peut jouer dans 0, un ou plusieurs concerts et un concert peut être composé d'un ou plusieurs artistes ;
- `Spectateur` : contient les informations sur les spectateurs ayant acheté des billets ainsi que leurs parrains éventuels. Un spectateur peut parrainer 0, un ou plusieurs autres spectateurs et un spectateur ne peut être le filleul d'au plus que d'un autre spectateur.
- `Billet` : relie chaque spectateur à un concert. Un spectateur peut participer à 0, un ou plusieurs concerts et un concert peut recevoir 0, un ou plusieurs spectateurs (il est annulé si aucun spectateur n'y participe). Un billet appartient à exactement un seul spectateur et un spectateur peut acheter 0, un ou plusieurs billets. Un billet est lié à exactement un concert et un concert à 0, un ou plusieurs billets qui lui sont rattachés (0 billet si aucun spectateur n'y participe auquel cas le concert sera annulé).

V.3.1. Convention Merise

Merise est le nom d'une méthodologie française apparue en France vers 1980. Elle a été développée par un groupe d'informaticiens et de chercheurs français, à la demande du ministère de l'Industrie (France), pour créer une méthode rigoureuse de conception des systèmes d'information (bases de données, logiciels, etc.).

Le nom « Merise » vient de l'acronyme « Méthode d'Étude et de Réalisation Informatique par les Sous-Ensembles ». Merise a repris les idées du modèle de Chen (que nous évoquerons par la suite), mais en l'adaptant sur certains points. La convention Merise est celle qui est classiquement enseignée en France.

Prenons cet exemple : (convention Merise)



Dans la **convention « Merise classique »** la cardinalité placée près de l'entité indique **combien de fois cette entité peut participer à l'association**. Chaque cardinalité est de la forme (min, max) :

- Côté « Ville » : (0,n) indique que la ville peut accueillir 0, un ou plusieurs Concert ;
- Côté « Concert » : (1,1) indique qu'un concert se déroulera dans une seule ville.

Voici quelques exemples supplémentaires avec la convention Merise (figure 44 ci-dessous) :

- un artiste participe à 0, un ou plusieurs concerts : (0,n) côté « Artiste » ;
- un concert peut recevoir un ou plusieurs artistes : (1,n) côté « Concert » ;
- un artiste a exactement une identité : (1,1) côté « Artiste » ;
- une identité est associée à exactement un artiste : (1,1) côté « Identité » ;
- un spectateur peut participer à 0, un ou plusieurs concerts : (0,n) côté « Spectateur » ;
- un concert peut accueillir 0, un ou plusieurs spectateurs : (0,n) côté « Concert » ;
- un spectateur peut parrainer 0, un ou plusieurs autres spectateurs : (0,n) entre Spectateur ↔ Parraine à gauche ;
- un spectateur a au plus un parrain : (0,1) entre Spectateur ↔ Parraine à droite ;

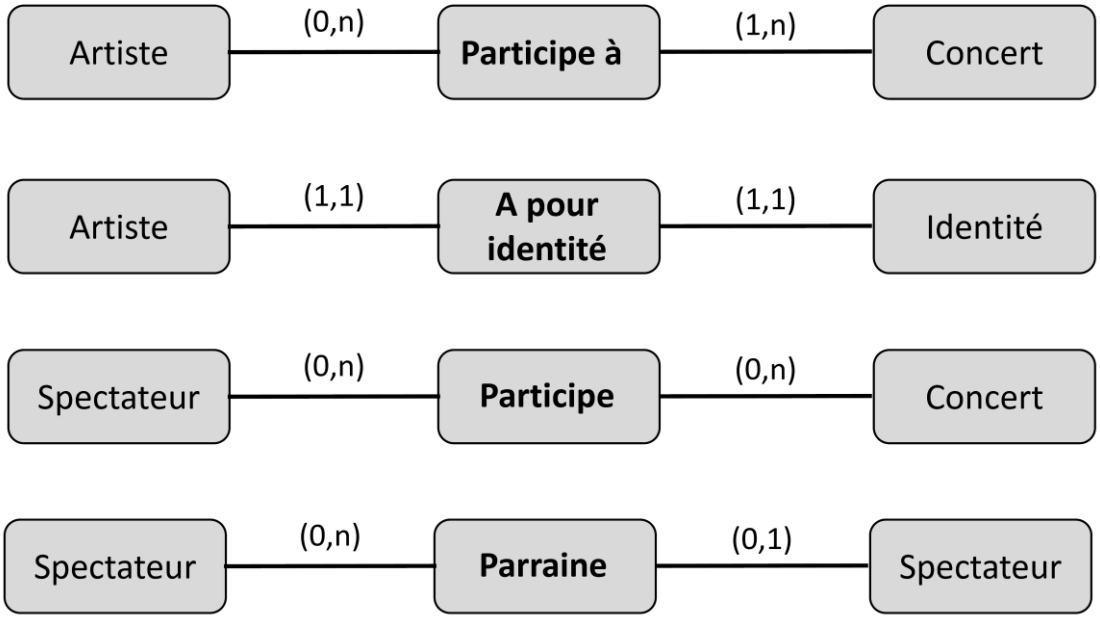


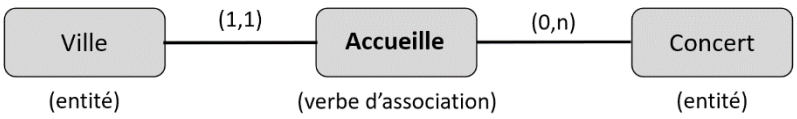
Figure 45 : Extrait du modèle entité - association de la base de données festival

V.3.2. Convention du modèle Chen

La convention Chen est plus internationale. Dans le modèle de Chen, les contraintes de cardinalités d'une entité donnée sont lues à partir des autres entités de l'association (sens entités connectées → entité concernée), alors qu'avec Merise, elles sont lues du sens entité concernée → entités connectées.

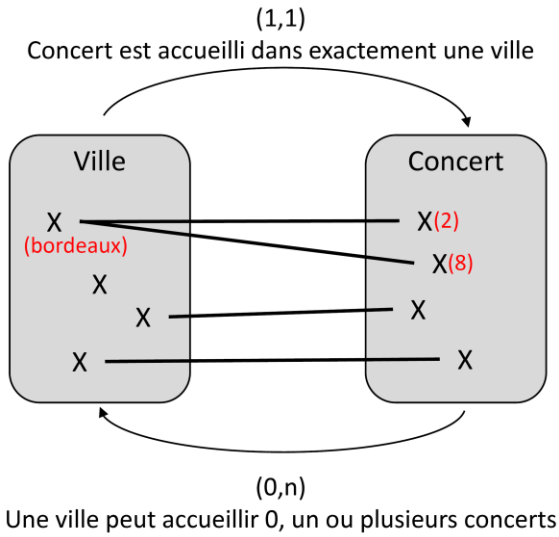
Ainsi, la cardinalité placée près de l'entité indique **combien de fois l'autre entité peut participer à l'association**.

Prenons cet exemple : (convention Chen)



- Côté « Ville » : (1,1) indique qu'un concert se déroulera dans une seule ville.
- Côté « Concert » : (0,n) indique qu'une ville peut accueillir 0, un ou plusieurs concerts.

Cela signifie dans notre base de données qu'une ville (comme Bordeaux – voir figure 45 ci-dessous) peut être reliée à plusieurs concerts (`id_concert` 2 et 8 : une même ville peut accueillir plusieurs concerts) et qu'à un `id_concert` spécifique doit obligatoirement être relié une et une seule ville (un concert ne se produit que dans une ville) :



Concert

id_concert	date	lieu	ville
Filtre	Filtre	Filtre	Filtre
2	2025-07-15	ParcBordeaux	Bordeaux
8	2025-08-02	ParcBordeaux	Bordeaux
5	2025-07-18	GrandPlace	Lille
10	2025-08-04	GrandPlace	Lille
4	2025-07-17	Zenith	Lyon
7	2025-08-01	Zenith	Lyon
3	2025-07-16	VieuxPort	Marseille
9	2025-08-03	VieuxPort	Marseille
1	2025-07-14	Hippodrome	Paris
6	2025-07-19	Hippodrome	Paris

Figure 46 : Correspondance entre la table Concert et le modèle EA

V.3.3. Convention UML

Le modèle UML (Unified Modeling Language) est un langage graphique unifié, créé dans les années 1990 pour décrire tout type de système informatique (en particulier les logiciels orientés objet).

Le modèle UML n'est pas une simple « évolution » directe de Chen, mais il hérite de ses idées, à travers plusieurs générations de méthodes intermédiaires (comme Merise). UML généralise le modèle de Chen à l'ensemble du système, pas seulement aux données. Chen a posé les bases de la modélisation des relations et des cardinalités, Merise les a formalisées pour les systèmes d'information, et UML les a intégrées dans un langage global de modélisation objet.

En UML, la convention est identique à celle de Chen et les cardinalités (0,1), (1,0), (1,1), (0,n), etc. sont remplacées par 0..1, 1..0, 1, 0..* etc. et sont appelées des **multiplicités**.

Cardinalités	Multiplicités UML
(0,1)	0..1
(1,1)	1 (ou absence)
(0,n)	0..* ou *
(1,n)	1..*

En UML, le verbe n'est pas obligatoire sur le lien de l'association.

Prenons l'exemple de l'auto-association `Spectateur` – `Spectateur` qui permet de mettre en œuvre le parrainage : un spectateur peut parrainer 0, un ou plusieurs autres spectateurs et un spectateur ne peut être le filleul d'au plus que d'un spectateur :

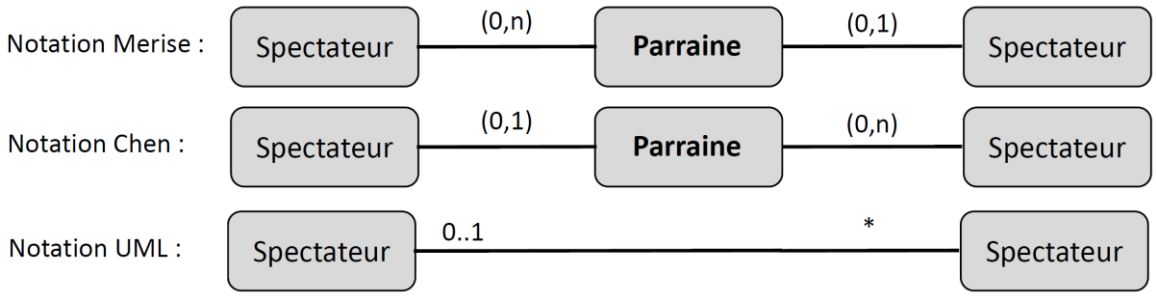


Figure 47 : Association Spectateur-Spectateur avec différentes conventions

La plupart du temps les types d'associations sont binaires (deux entités reliées par un lien) et appartiennent à une des trois classes fonctionnelles suivantes : 1 – 1 (associations un-à-un), 1 – * (ou * – 1, associations un-à-plusieurs), ou * – * (associations plusieurs-à-plusieurs) :

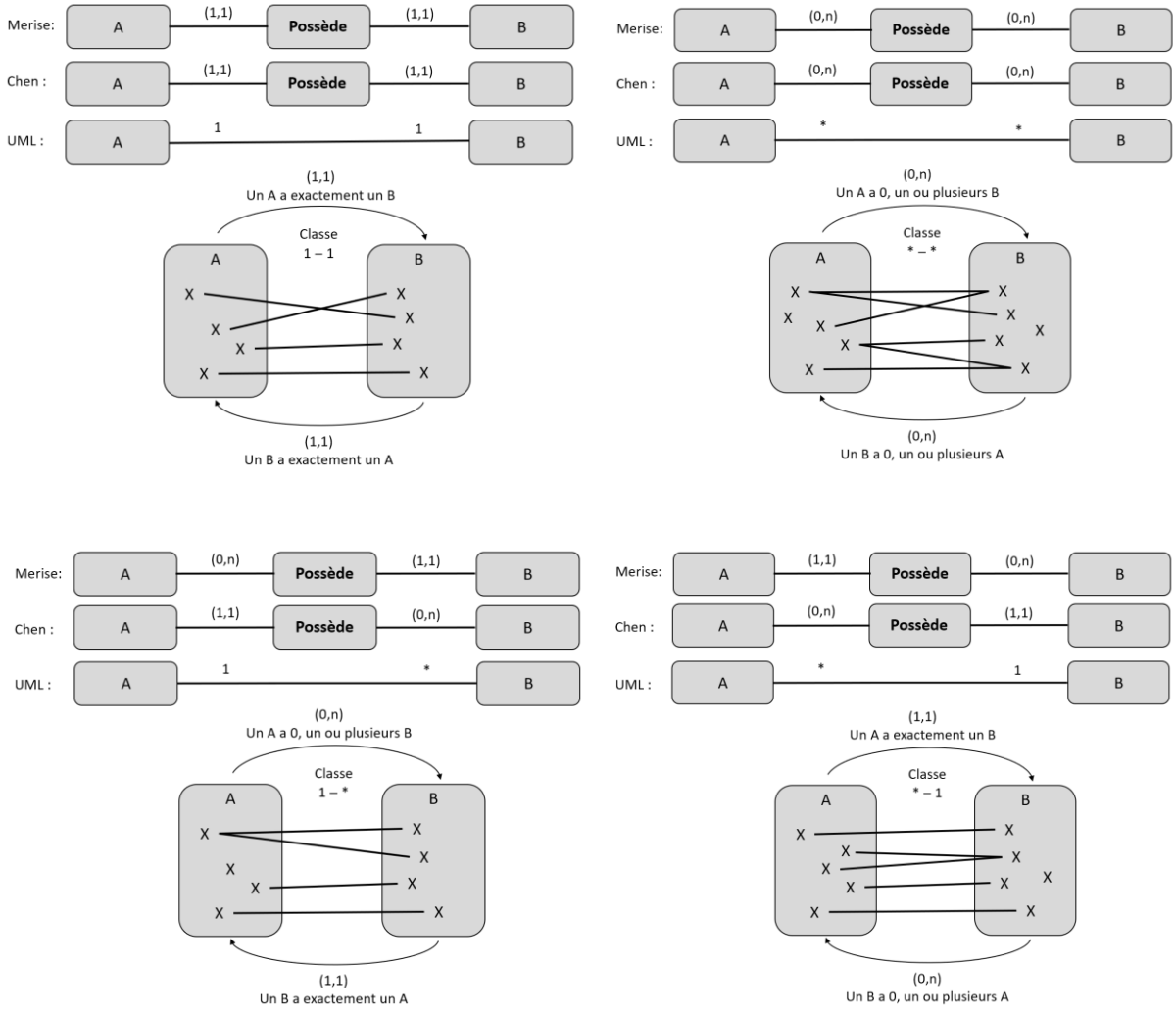


Figure 48 : Illustration des principales classes : 1-1, 1-* (ou *-1) et *-*

VI) LE MODÈLE RELATIONNEL (SQL)

Après avoir représenté le système d'information à l'aide du modèle entité–association, nous disposons désormais d'une vision claire et logique des entités du monde réel et des relations qui les unissent.

Mais pour qu'une base de données puisse être exploitée par un logiciel comme un SGBD (Système de Gestion de Base de Données), il faut traduire ce modèle conceptuel en une forme opérationnelle et structurée : le modèle relationnel.

Ce modèle, utilisé par les langages tels que SQL, ne travaille plus avec des entités et des verbes, mais avec des tables, des colonnes et des contraintes. Chaque entité du modèle EA devient une table, chaque association se traduit par une clé ou une table de liaison, et les cardinalités sont mises en œuvre à travers les contraintes d'intégrité du SGBD.

Le modèle relationnel constitue la mise en œuvre technique du modèle conceptuel : il transforme les liens logiques du modèle entité–association en structures physiques que SQL peut manipuler, interroger et contrôler.

VI.1. De l'entité à la table

Chaque entité du modèle EA peut être représenté par une **table**, qui porte le nom du **type d'entité** ou par un attribut d'une table. On comprend au passage pourquoi les noms des tables sont au singulier (Artiste par exemple, et pas Artistes, alors même qu'une table de ce genre représente a priori un ensemble d'artistes) : le point de vue est celui du modèle EA (Entité-Association) : une table correspond à un type d'entité (au singulier) plus qu'à un ensemble d'instances (au pluriel).

VI.2. De la cardinalité conceptuelle à la structure relationnelle

Lorsqu'on passe du modèle entité–association (EA) au modèle relationnel (SQL), il ne s'agit pas simplement de changer de langage : on change aussi de niveau d'abstraction.

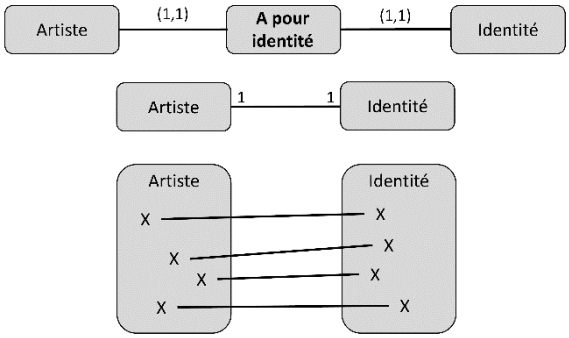
Dans le modèle EA, les cardinalités – notées (1,1), (0,1), (1,n), (0,n) — décrivent des contraintes logiques entre les entités du monde réel : combien d'occurrences d'une entité peuvent être associées à combien d'occurrences d'une autre.

Dans le modèle SQL, ces mêmes liens doivent être traduits techniquement. On ne parle plus de « combien d'éléments peuvent être reliés » d'un point de vue conceptuel, mais de comment ces liens sont mis en œuvre techniquement.

Ainsi, en SQL on n'utilise plus de verbes (« participe à ») ni de valeurs minimales et maximales, mais à la place, on indique comment les liens entre les tables vont être mis en œuvre soit à l'aide de clés associées à leurs contraintes (si elles sont obligatoires ou optionnelles) ou de tables associatives.

VI.2.1. La classe 1–1

Prenons l'association Artiste – Identité dans le modèle Merise, qui est de type (1,1) ↔ (1,1) :



C'est un lien exclusif : aucune identité sans artiste, aucun artiste sans identité.

En SQL, cette relation est implantée en utilisant une clé primaire d'un côté et une clé étrangère de l'autre.

Dans notre base de données, la clé primaire est du côté de la table `Artiste` et la clé étrangère du côté de la table `Identite` (mais cela aurait pu être l'inverse) :

Clé primaire

Artiste

id_artiste	nom	pays	style
Filtre	Filtre	Filtre	Filtre
1	Aya Nakamura	France	pop
2	Stromae	Belgique	pop
3	Daft Punk	France	electro
4	Angèle	Belgique	pop
5	Orelsan	France	rap
6	PNL	France	rap
7	Jul	France	rap
8	David Guetta	France	electro
9	Jain	France	pop
10	Kendrick Lamar	USA	rap
11	Coldplay	UK	rock
12	Muse	UK	rock

Clé étrangère

Identité

id_identite	id_artiste	numero_secu
Filtre	Filtre	Filtre
1	1	1840575123456
2	2	2870415234567
3	3	1900123456789
4	4	2930815234598
5	5	1790512234567

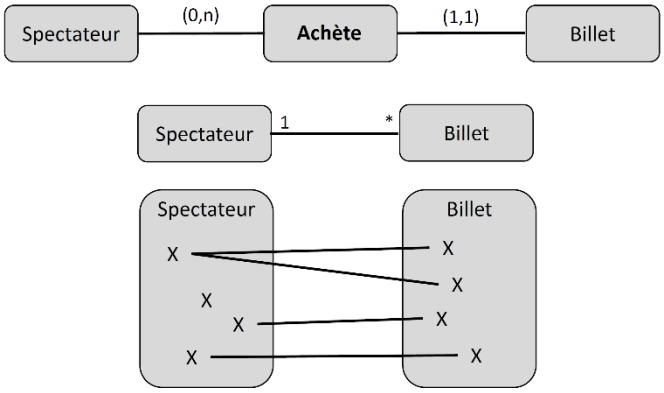
Identité(id_identite, id_artiste, numero_secu)

Artiste(id_artiste, nom, pays, style)

Figure 49 : Implantation de la relation 1-1 entre Artiste et Identité dans la base de données

*VI.2.2. La classe 1–**

Prenons l'exemple de l'association `Spectateur` – `Billet` dans le modèle Merise, qui est de type (0,n) ↔ (1,1). Dans cette configuration, on utilise une clé étrangère du côté « * » (dans `Billet`).



Utiliser une clé étrangère dans la table `Spectateur` est impossible car il faudrait alors insérer plusieurs valeurs dans une cellule de cette clé étrangère.

Cela violerait la contrainte que les cellules doivent contenir des valeurs atomiques.

On voit sur la figure 48 ci-dessous que la clé étrangère est dans `Billet` :

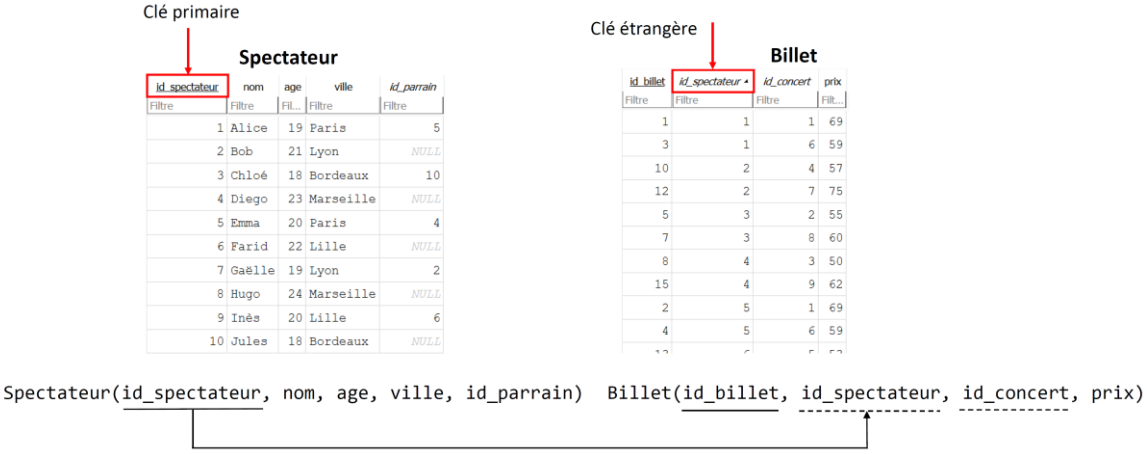
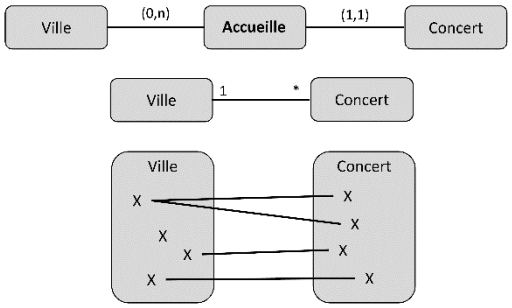


Figure 50 : Implantation de la relation 1-* entre Spectateur et Billet dans la base de données

Cette manière de réaliser l'implantation de la relation 1–* est un cas standard. C'est une solution flexible car un spectateur peut avoir 0, 1 ou plusieurs billets et les billets peuvent contenir des attributs propres (`prix`, `id_concert`).

Mais on peut également la réaliser cette association avec une seule table si la relation est très contrainte et ne demande pas autant de flexibilité. C'est le cas dans l'association `Ville` – `Concert` dans notre base de données (convention Merise sur le schéma du haut) :

id_concert	date	lieu	ville
2	2025-07-15	ParcBordeaux	Bordeaux
8	2025-08-02	ParcBordeaux	Bordeaux
5	2025-07-18	GrandPlace	Lille
10	2025-08-04	GrandPlace	Lille
4	2025-07-17	Zenith	Lyon
7	2025-08-01	Zenith	Lyon
3	2025-07-16	VieuxPort	Marseille

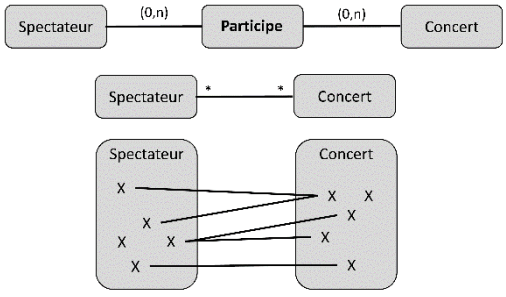


Cependant, cette implantation ne donne aucune flexibilité : il est impossible de gérer plusieurs entités ou d'ajouter des attributs.

*VI.2.3. La classe *–*, décomposition en 1–* et table de liaison*

Intéressons-nous à la classe * – * avec l'association `Artiste` – `Concert`.

Dans cette configuration, on veut que plusieurs spectateurs puissent être en relation avec plusieurs concerts et inversement.



Il faudrait donc pour cela que la clé primaire d'une des deux tables et la clé secondaire de l'autre puissent enregistrer plusieurs valeurs identiques par clé simultanément. Cela serait contradictoire avec la spécificité des clés primaires car elles ne doivent contenir que des valeurs uniques.

La solution est d'utiliser une **table de liaison** `Billet` : pour exprimer ce lien plusieurs-à-plusieurs, on ne touche pas aux clés primaires des tables `Spectateur` et `Concert` et ces deux clés primaires pointent vers deux clés secondaires de la table de liaison `Billet`. Comme ce sont des clés secondaires, elles peuvent enregistrer des valeurs non uniques.

Cela permet de décomposer la relation * – * en deux relations 1 – * (les deux schémas au-dessus utilisent convention Merise et le schéma du bas la convention UML) :

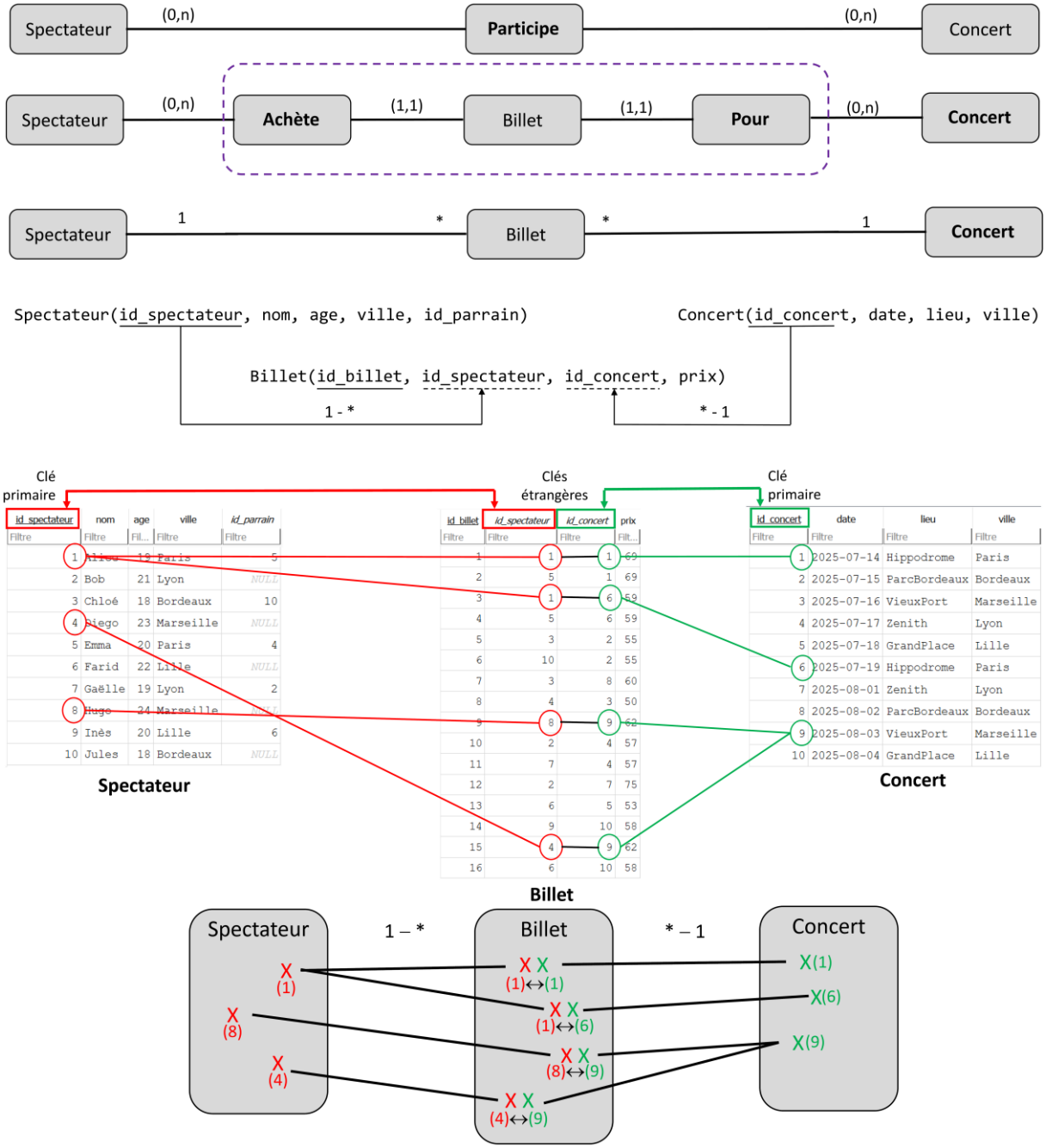


Figure 51 : Décomposition d'une classe *-* en deux classes 1-*

Ainsi, dans l'exemple illustré en figure 50, on voit par exemple qu'un spectateur (1) est lié à plusieurs concerts (1 et 6) et qu'un concert (9) est lié à plusieurs spectateurs (4 et 8).

En SQL, une relation plusieurs-à-plusieurs (* – *) n'existe pas directement et on la décompose toujours en deux relations 1 – * via une table intermédiaire (souvent appelée table de liaison) :

- Côté `Spectateur` : [1 – *] – Un spectateur peut être lié à plusieurs billets et un billet appartient exactement à un seul spectateur.
- Côté `Concert` : [1 – *] vu de `Concert` – Un concert peut être lié à plusieurs billets et un billet ne donne droit qu'à un seul concert.

En conclusion, une association de type * – * ne peut pas être représentée directement dans une base relationnelle. Elle est décomposée en deux associations 1 – * grâce à une table d'association. Cette table contient les clés étrangères des deux entités principales.