Requêtes en langage SQL

Dans le chapitre précédent, nous avons découvert la manière dont les données sont organisées dans une base de données relationnelle (BDDR). Pour interroger une telle base et en extraire de l'information, on code des *requêtes* dans un langage particulier appelé SQL pour *Structured Query Language* (langage structuré de requêtes). Ce langage est régi par une norme internationale et a été adopté par tous les grands éditeurs de logiciels, de sorte qu'on peut l'utiliser avec presque tous les SGBDR (systèmes de gestion de bases de données relationnelles), tels que Oracle Database, Microsoft SQL Server, MySQL, SQLite, etc.

Nous n'allons étudier qu'une petite partie de ce langage en nous concentrant sur la commande `SELECT` et ses nombreuses variantes. L'utilisation de chaque mot-clé sera illustrée par des exemples portant le plus souvent sur la BDD `modeles_reduits` décrite en annexe et dont il est nécessaire de prendre connaissance avant d'entrer dans le vif du sujet. Je vous recommande de dégrafer la page correspondante pour la consulter plus facilement.

Toutes les exécutions se feront dans l'environnement de développement SQLite Studio, qui permet la consultation visuelle du contenu de la base et l'affichage des résultats dans une fenêtre dédiée. Tout au long du cours, c'est le professeur qui procédera aux manipulations, puis vous devrez le faire vous-mêmes en TD. À titre d'illustration, le résultat de certaines requêtes a été imprimé dans ce document, mais seulement en partie pour limiter la consommation de papier. Vous pourrez bien entendu reprendre les exemples sur machine lors de l'apprentissage de votre cours.

I Extraction d'information d'une table

On aborde dans cette partie les formes les plus simples de la requête `SELECT`, dans lesquelles l'information est extraite d'une seule table de la BDD.

I.1 Extraction de colonnes

La requête SQL la plus simple consiste à récupérer certaines colonnes d'une table. Cette opération se nomme « *projection* » et se code comme suit.

```
1 SELECT attribut1, attribut2, ... FROM une_table
```

Voici un exemple d'exécution et un extrait du résultat, dans lequel les lignes apparaissent dans un ordre quelconque. Il faut comprendre que la requête renvoie un nouvelle table et, comme nous l'avons souligné dans le chapitre précédent, il n'existe pas de concept d'ordre dans une table.

```
1 SELECT nom_client, ville FROM client
```

nom_client	**ville**
Atelier graphique	Nantes
Signal Gift Stores	Las Vegas
La Rochelle Gifts	Nantes
Baane Mini Imports	Stavern
Mini Gifts Distributors Ltd.	San Rafael
Havel Zbyszek Co	Warszawa
Daedalus Designs Imports	Lille
La Corne D'abondance	Paris

Pour sélectionner toutes les colonnes d'une table, on utilise le caractère « * » qui joue le rôle de joker.

```
1 SELECT * FROM commande
```

num_commande	code produit	quantite	prix unitaire	ligne bon commande
10100	S18 1749	30	136.0	3
10100	S18 2248	50	55.09	2

Par défaut, chaque colonne du résultat porte le nom de l'attribut correspondant dans la table originale. Afin de rendre le résultat plus lisible (ou de le réutiliser dans la suite la même requête), on peut renommer certaines colonnes avec le mot-clé `AS`. Une requête peut aussi calculer le résultat de diverses opérations arithmétiques (+, -, *, /) faisant intervenir des attributs numériques[1]. Dans ce cas, le renommage s'avère particulièrement pertinent.

1. Certains SGBD permettent de plus l'utilisation des fonctions mathématiques courantes.

Observer l'exemple suivant et remarquer qu'un nom de variable SQL peut comporter des espace, à condition de l'encadrer par des guillemets doubles.

```
1 SELECT nom_article AS modèle, prix_vente_recommandé - prix_achat AS "Bénéfice␣en␣
      euros" FROM article;
```

modèle	Bénéfice en euros
1969 Harley Davidson Ultimate Chopper	46.89
1952 Alpine Renault 1300	115.72
1996 Moto Guzzi 1100i	49.95
2003 Harley-Davidson Eagle Drag Bike	102.64
1972 Alfa Romeo GTA	50.32

Procédons maintenant à quelques applications élémentaires de ces notions. Coder les requêtes répondant aux questions suivantes.

- Afficher les colonnes `ville` et `pays` de la table `bureau`.

- Afficher toutes les colonnes de la table `article`.

- Pour chaque ligne de la table `detail_commande`, afficher le numéro de la commande, la ligne correspondant sur le bon de commande en la renommant « ligne » et le prix total payé par le client pour cette ligne en le renommant « prix payé ».

I.2 Extraction de colonnes avec sélection de lignes

On peut compléter la requête `SELECT ... FROM` de manière à ne conserver que les lignes satisfaisant une certaine condition, selon une opération nommée « *sélection* ». La requête s'écrit alors sous la forme suivante.

```
1 SELECT attribut1, attribut2, ... FROM une_table WHERE condition
```

La condition de sélection peut comporter les tests d'égalité (`=`, et non pas `==` comme en Python), de différence (`!=`, que l'on peut aussi écrire `<>`) et de comparaison (`<`, `<=`, `>`, `>=`). On peut combiner plusieurs tests avec les opérateurs `AND`, `OR`, `NOT` (pour la négation). D'un point de vue logique, le résultat d'un test est bien entendu une grandeur booléenne, mais il n'existe pas à proprement parler de type booléen en SQL [2]. Les valeurs notées `TRUE` et `FALSE` en Python sont représentées par les entiers 1 et 0.

Il est à noter que les opérateurs de comparaison s'appliquent aussi aux chaînes de caractères, l'ordre utilisé étant celui du dictionnaire (ordre lexicographique). Par exemple, `'chat' < 'chien'` est vrai alors que `'elephant' > 'souris'` est faux. Il est bon de s'en convaincre en exécutant en exemples quelques requêtes de la forme `SELECT chaine1 < chaine2`.

```
1 SELECT 4 < 12 ; -- renvoie 1
  SELECT '4' < '12' ; -- renvoie 0. À expliquer !
```

Vous remarquerez à cette occasion que les chaînes doivent être délimitées par des guillements simples. Si cette chaîne contient elle-même des guillements, il faudra les redoubler, en écrivant par exemple `'ajourd''hui'` pour obtenir « aujourd'hui ».

Reprenons l'exemple de la BDD `modeles_reduits`. Les échelles de reproduction des différents véhicules sont représentées par des chaînes de caractères telles que `'1:10'`. La comparaison de deux telles chaînes permet de savoir, parmi deux véhicules, lequel est reproduit avec l'échelle de réduction la plus forte (par exemple, `'1:12' < '1:18'` est vrai, la seconde échelle de réduction est plus forte). Malheurseument, cela ne marche pas si le second

2. Les mot-clés `TRUE` et `FALSE` sont cependant reconnus, mais il faut les considérer comme des réécritures de 1 et 0

nombre est écrit avec trois chiffres : on a `'1:700' < '1:72'` alors que 700 > 72. On voit ici l'inconvénient d'avoir typé les échelles comme des chaînes de caractères.

Si nous devons comparer des dates, nous les traiterons comme des chaînes de caractères ou comme des valeurs numériques. Si chaque date est écrite sous la forme année-mois-jour 'AAAA-MM-JJ', la comparaison lexicographique des chaînes de caractères coïncide avec la comparaison chronologique des dates. On peut le vérifier en exécutant les deux requêtes `SELECT '2023-11-04' > '2023-05-04'` et `SELECT '2021-31-12' < '2022-01-01'` qui renvoient toutes deux la valeur « booléenne » 1.

Passons à quelques exemples de sélection. La requête suivante renvoie le nom de chacun des employés du bureau portant le code 6.

```
1 SELECT nom FROM employe WHERE code_bureau = 6
```

Expliquer ce que font les requêtes suivantes.

```
1 SELECT nom_article FROM article WHERE categorie_article = 'Motorcycles' AND stock >
     5000  ;
```

```
1 SELECT nom_article, echelle, prix_achat, prix_vente_recommandé,
     100*(prix_vente_recommandé - prix_achat)/prix_achat AS marge, '%'
  FROM article
  WHERE categorie_article = 'Planes'
4 AND marge > 60 ;
```

Codons maintenant les requêtes répondant aux questions suivantes.

- Afficher les villes dans lesquelles sont localisés tous les bureaux des USA.

- Trouver le nom de tous les articles reproduits avec une réduction plus forte que 20 et disponibles chez d'autres fournisseurs que `Carousel DieCast Legends`. Afficher le nom de l'article, celui du fournisseur et l'échelle.

- Trouver les numéros de toutes les commandes passées en 2003 pour une livraison avant le jour de Noël et qui n'ont pas été annulées (`'Cancelled'`).

- Trouver le numéro de toutes les commandes comportant l'article `S18_1749` en au moins 30 exemplaires et afficher ce nombre.

I.3 Organiser les résultats d'une requête

a. Mot-clé DISTINCT

Une requête `SELECT` renvoie autant de lignes qu'il y en a, dans la table originale, satisfaisant la condition spécifiée après `WHERE`. Ce comportement ne répond pas toujours au besoin de l'utilisateur. Si on veut connaître les différentes catégories d'articles, la requête ci-dessous ne convient pas ; elle donne un résultat de 110 lignes dans lequel les catégories identiques se répètent inutilement.

categorie article
Motorcycles
Classic Cars
Motorcycles
Motorcycles
Classic Cars
Classic Cars

```sql
SELECT categorie_article FROM article
```

Pour l'éviter, on utilise le mot-clé `DISTINCT`, suivi du nom d'un attribut, qui élimine toutes les répétitions. On peut alors lire commodément les 7 catégories de véhicules miniatures disponibles.

categorie article
Motorcycles
Classic Cars
Trucks and Buses
Vintage Cars
Planes
Ships
Trains

```sql
SELECT DISTINCT categorie_article FROM article
```

Le caractère distinct de deux lignes peut aussi être évalué en comparant plusieurs de leurs attributs. Dans le cas de deux attributs A_1 et A_2, les lignes t et u sont considérées identiques si A_1 *et* A_2 prennent les mêmes valeurs sur ces deux lignes, et au contraire distinctes si *l'un au moins* des deux attributs diffère d'une ligne à l'autre. Ainsi, la requête ci-dessous permet d'obtenir sans répétition tous les couples (categorie, echelle), avec l'idée qu'une voiture d'échelle `1:12` n'est pas interchangeable avec une voiture d'échelle `1:24`.

```sql
SELECT DISTINCT categorie_article, echelle FROM article
```

Bien entendu, la clause `DISTINCT` peut se combiner avec `WHERE`. Par exemple, pour trouver tous les numéros de commande dans lesquels au moins un article de plus de 50 € a été vendu en plus que 50 exemplaire, sans répétition, on utilise la requête ci-dessous. Sans le mot-clé `DISTINCT`, on aurait répété les commandes pour lesquelles plusieurs articles satisfont la condition.

```sql
SELECT DISTINCT num_commande FROM detail_commande
WHERE prix_unitaire > 50 AND quantite > 50
```

À vous maintenant de coder les requêtes répondant aux questions suivantes.

1. Obtenir la liste sans répétition des pays de résidence des clients.

2. Trouver les numéros des commandes pour lesquelles le bon de commande comporte au moins 3 lignes (c'est à dire pour lesquelles une ligne avec un numéro supérieur ou égal à trois existe).

3. Trouver les numéros sans répétition des clients ayant procédé à au moins un paiement de moins de 20000€.

b. Mot-clé ORDER BY

On peut trier les lignes résultant d'une requête `SELECT` grâce au mot-clé `ORDER BY`, suivi du nom d'un attribut sur lequel existe une relation d'ordre. Par défaut, l'ordre croissant (`ASC`) est utilisé et l'ordre décroissant s'obtient par le mot-clé `DESC`.

```sql
SELECT nom_article,prix_achat FROM article ORDER BY prix_achat
```

nom article	prix achat
1958 Chevy Corvette Limited Edition	15.91
1982 Lamborghini Diablo	16.24
1938 Cadillac V-16 Presidential Limousine	20.61
1936 Mercedes Benz 500k Roadster	21.75

L'attribut utilisé pour déterminer l'ordre des lignes peut ne pas figurer parmi ceux à extraire. On peut ainsi obtenir la date des différents paiements en les triant selon leur montant, par ordre décroissant.

```
SELECT date_paiement FROM paiement ORDER BY montant DESC
```

Lorsque plusieurs attributs entrent en jeu dans le tri, on classe d'abord selon le premier, puis en cas d'égalité selon le second, etc. Il est aussi possible d'utiliser un alias (un renommage) d'attribut.

```
SELECT code_article, echelle, prix_vente_recommandé AS PVR FROM article
ORDER BY echelle, PVR
```

Conjointement au mot clé `ORDRE BY`, on peut utiliser `LIMIT N_1 OFFSET N_2` pour indiquer qu'on souhaite sélectionner seulement les N_1 premières lignes du classement. La partie `OFFSET N_2`, facultative, indique que l'on doit sauter les N_2 premières lignes, puis sélectionner seulement les N_1 qui suivent. Ainsi, la requête suivante affiche les lignes de bons de commande occupant les places 21 à 25 dans le palmarès établi selon la quantité de chaque article.

```
SELECT num_commande, quantite FROM detail_commande
ORDER BY quantite DESC LIMIT 5 OFFSET 20
```

Rien n'interdit d'utiliser `DISTINCT` simultanément à `ORDER BY`. Dans ce cas, les lignes sont triés *après* que les doublons ont été éliminés. À ce sujet, comparer le résultat de la requête suivante à celui que l'on obtient si on en supprime la clause `DISTINCT`.

```
SELECT DISTINCT quantite FROM detail_commande
ORDER BY quantite DESC LIMIT 5 OFFSET 20 ;
```

1. Afficher le nom de tous les clients ayant une limite de crédit supérieure à 50000 €, leur pays et leur ville en les triant par ordre alphabétique de leur pays et, en cas d'égalité, par ville.

2. Afficher intégralement le détail des commande en y ajoutant le chiffre d'affaire correspondant, que l'on renommera `CA`, et les trier selon ce chiffre.

3. Afficher le nom et la limite de crédit du client qui a, parmi ceux résidant aux USA et à qui on accorde un crédit non nul, le 20e plus petit crédit.

II Opérations ensemblistes

Le résultat d'une requête est une nouvelle table qu'on peut considérer, d'un point de vue mathématique, comme un *ensemble* d'enregistrements. En tant que tel, elle peut intervenir dans les opérations ensemblistes usuelles que sont l'union, l'intersection, la différence et le produit cartésien. On peut effectuer les trois premières opérations avec les mot-clés `UNION`, `INTERSECTION` et `EXCEPT`, *à condition de faire intervenir des tables possédant le même nombre de colonnes et avec des attributs du même type.*

II.1 Union, intersection, différence

Pour trouver les employés travaillant dans le bureau 1 ou dans le bureau 2, utilisons une union.

```
SELECT nom, prenom FROM employe WHERE code_bureau = 1
```

```
UNION
SELECT nom, prenom FROM employe WHERE code_bureau = 2 ;
```

Pour trouver ceux qui travaillent dans le bureau 6 et dont le superviseur est l'employé 1088, utilisons une intersection.

```
SELECT nom, prenom FROM employe WHERE code_bureau = 6
INTERSECT
SELECT nom, prenom FROM employe WHERE superviseur = 1088
```

Enfin, pour trouver ceux qui travaillent dans le bureau 6 mais dont le superviseur n'est pas l'employé 1088, utilisons une différence.

```
SELECT nom, prenom FROM employe WHERE code_bureau = 6
EXCEPT
SELECT nom, prenom FROM employe WHERE superviseur = 1088 ;
```

Dans ces trois exemples, on obtient le même résultat en remplaçant l'opération ensembliste un opérateur logique `OR`, `AND` ou `AND NOT` respectivement, placé dans la clause `WHERE`.

```
SELECT nom, prenom FROM employe WHERE code_bureau = 1 OR code_bureau = 2

SELECT nom, prenom FROM employe WHERE code_bureau = 6  AND superviseur = 1088

SELECT nom, prenom FROM employe WHERE code_bureau = 6 AND NOT superviseur = 1088
```

Ces exemples peuvent laisser penser qu'il est superflu d'apprendre à coder des opérations ensemblistes. Elles trouvent pourtant un intérêt lorsqu'on les applique à des résultats de requêtes portant sur des tables distinctes [3]. Cherchons par exemple les villes et pays possédant soit un client, soit un bureau, voire les deux.

```
SELECT DISTINCT ville, pays FROM client
UNION
SELECT DISTINCT ville, pays FROM bureau
```

Noter que le mot-clé `DISTINCT` est superflu parce que l'opérateur `UNION` supprime automatiquement d'éventuels doublons présents dans la réunion des deux tables.
Cherchons maintenant les villes qui possèdent à la fois au moins un client et un bureau.

```
SELECT DISTINCT ville FROM client
INTERSECT
SELECT DISTINCT ville FROM bureau
```

Et enfin, les villes qui possèdent au moins un client mais aucun bureau.

```
SELECT DISTINCT ville FROM client
EXCEPT
SELECT DISTINCT ville FROM bureau
```

II.2 Produit cartésien

Commençons par un rappel de mathématiques. Le produit cartésien $A \times B$ de deux ensembles A et B est l'ensemble de tous les couples (x, y) avec $x \in A$ et $y \in B$. Si A et B sont dénombrables, le cardinal $A \times B$ est le produit du cardinal de A par celui de B.
Soient deux tables T_1 et T_2 possédant respectivement N_1 et N_2 enregistrements. Le produit cartésien de T_1 et T_2 est la table formée des lignes obtenues en concaténant chaque ligne de T_1 avec chaque ligne de T_2. Elle possède $N_1 \times N_2$ enregistrements. Par exemple, le produit cartésien de la table `employe` par la table `bureau` comporte $23 \times 7 = 161$ lignes et 13 colonnes. En SQL, ce produit cartésien ne s'obtient pas par le symbole $\times$ comme en mathématiques, mais simplement en plaçant l'un après l'autre les noms des tables, séparés d'une virgule.

```
SELECT * FROM employe, bureau
```

3. L'intersection pourrait être remplacée par une jointure, comme nous le verrons plus loin.

Bien entendu, il est possible de ne retenir que certaines colonnes de ce produit cartésien.

```
1 SELECT nom, prenom, code_bureau as codbur, superviseur AS superv,
  fonction, code, ville, telephone, adresse, pays
  FROM employe, bureau
```

Quelques lignes extraites du résultat sont imprimées ci-dessous

nom	prenom	code bureau	superviseur	code	ville	adresse	pays
Nishi	Mami	5	1056	1	San Francisco	100 Market Street	USA
Nishi	Mami	5	1056	4	Paris	43 Rue Jouffroy D'abbans	France
Nishi	Mami	5	1056	5	Tokyo	4-1 Kioicho	Japan
Kato	Yoshimi	5	1621	1	San Francisco	100 Market Street	USA
Kato	Yoshimi	5	1621	4	Paris	43 Rue Jouffroy D'abbans	France
Kato	Yoshimi	5	1621	5	Tokyo	4-1 Kioicho	Japan
Gerard	Martin	4	1102	1	San Francisco	100 Market Street	USA
Gerard	Martin	4	1102	4	Paris	43 Rue Jouffroy D'abbans	France
Gerard	Martin	4	1102	5	Tokyo	4-1 Kioicho	Japan

Dès que les tables sont un peu longues, ce produit cartésien occupe beaucoup d'espace en mémoire, même s'il n'existe que durant le temps de la requête. Par ailleurs, on peut se demander quel intérêt il y a à associer aveuglément chaque enregistrement d'une table à chaque enregistrement de l'autre. À quoi bon écrire sur la même ligne ce qui concerne l'employé Gérard Martin d'une part et ce qui concerne le bureau de Tokyo d'autre part ? À vrai dire, le produit cartésien ne présente pas un grand intérêt pratique. Nous l'introduisons surtout en vue de mieux comprendre la partie suivante concernant les jointures.

Voici tout de même un exemple d'utilisation du produit cartésien. La requête suivante trouve tous les couples (article, client) tel que le client soit en mesure d'acheter à crédit tout le stock disponible pour cet article selon son prix de vente recommandé (il faut pour cela que sa limite de crédit soit supérieure à la valeur du stock pour cet article).

```
1 SELECT code_article, nom_client
2 FROM article, client
  WHERE prix_vente_recommandé * stock < limite_de_credit ;
```

III Jointures

III.1 Principe

Reprenons l'exemple précédent. Plutôt que de coupler aveuglément chaque ligne de la table `employe` à chaque ligne de la table `bureau`, il serait plus pertinent d'afficher, à côté du nom de chaque employé, les informations concernant le bureau *dans lequel il travaille réellement*. L'attribut `code_bureau` dans la table `employe` constitue une clé étrangère vers la table `bureau` et a précisément pour fonction de permettre la passage raisonné d'une table à l'autre ; il exprime l'association un-à-plusieurs `travaille dans le`. Pour exprimer cette idée en SQL, on utilise un produit cartésien combiné à une clause `WHERE` sélectionnant exclusivement les lignes pertinentes.

```
1 SELECT nom, prenom, code_bureau, code, ville, pays FROM employe, bureau WHERE
     code_bureau = code
```

nom	prenom	code bureau	code	ville	pays
Thompson	Leslie	1	1	San Francisco	USA
Bondur	Loui	4	4	Paris	France
Nishi	Mami	5	5	Tokyo	Japan
Gerard	Martin	4	4	Paris	France
Patterson	Mary	1	1	San Francisco	USA
Castillo	Pamela	4	4	Paris	France
Marsh	Peter	6	6	Sydney	Australia

Nous voyons ainsi que Nishi Mami travaille à Tokyo et Gerard Martin à Paris.

On appelle *jointure entre tables* cette combinaison d'un produit cartésien et d'une sélection selon une condition

faisant intervenir les colonnes de deux tables. Dans les limites de notre programme, il s'agit d'exprimer l'égalité entre la clé primaire de l'une des table et une clé étrangère de l'autre.

Dans une BDDR, l'information est répartie dans plusieurs tables conformément au modèle entité-association étudié dans le chapitre précédent. Le système de clés exprime les liens logiques qu'entretiennent les différentes entités les unes avec les autres. L'utilisation des jointures permet d'exploiter ces liens logiques lors de la recherche d'information, en sollicitant plusieurs tables au sein d'une même requête. Ce mode de fonctionnement est à la base de l'efficacité du modèle relationnel.

III.2 Codage des jointures

Plutôt qu'un produit cartésien combiné à une clause `WHERE`, le programme officiel nous invite à utiliser la syntaxe suivante pour les jointures.

```sql
SELECT attibuts FROM table1 JOIN table2 ON condition
```

La recherche du bureau d'affectation de chaque employé s'écrira donc comme suit. On constate qu'il n'est pas nécessaire d'extraire les colonnes sur lesquelles porte la jointure.

```sql
SELECT nom, prenom, ville, pays FROM employe JOIN bureau ON code_bureau = code
```

Voyons un autre exemple. À chaque client (table `client`) est associé un employé (table `employe`) responsable des ventes, l'association se faisant par la clé étrangère `resp_ventes`. Cherchons le nom et le prénom du responsable des ventes de chaque client.

De même que le produit cartésien, la jointure peut faire intervenir plus de deux tables. Cherchons, en plus du nom et du prénom, le numéro de téléphone du bureau auquel est rattaché le responsable des ventes de chaque client.

```sql
SELECT nom_client, nom, prenom, bureau.telephone
FROM client JOIN employe ON res_ventes = num_employe JOIN bureau ON code_bureau =
    code ;
-- ou bien
SELECT nom_client, nom, prenom, bureau.telephone
FROM bureau JOIN employe ON code_bureau = code  JOIN client ON res_ventes =
    num_employe ;
```

Noter que les deux requêtes conduisent au même résultat affiché ci-dessous : l'ordre des jointures est indifférent. Noter aussi l'utilisation du préfixe « bureau » qui évite l'ambiguïté sur l'attribut « telephone » : on veut le numéro de téléphone dans la table « bureau » et non pas dans la table « client ».

nom client	nom	prenom	telephone
Atelier graphique	Hernandez	Gerard	+33 14 723 4404
Signal Gift Stores	Thompson	Leslie	+1 650 219 4782
La Rochelle Gifts	Hernandez	Gerard	+33 14 723 4404
Baane Mini Imports	Jones	Barry	+44 20 7877 2041

Lorsqu'on manipule de nombreuses colonnes, il devient difficile de se rappeler le nom de chacune et on oublie même parfois à quelle table certains attributs appartiennent. De plus, des problèmes d'homonymie entres les attributs apparaissent fréquemment. Par exemple, il existe un attribut `ville` dans la table `bureau` et un autre attribut `ville` dans la table `client`. Pour éviter les confusions, on fait précéder le nom de chaque attribut de celui de la table d'où il est extrait, selon la syntaxe `table.attribut`.

```sql
SELECT client.nom_client, client.ville, bureau.telephone, bureau.ville
FROM bureau JOIN employe ON code_bureau = code
JOIN client ON res_ventes = num_employe
```

Il est aussi permis de procéder à un renommage de table. Remarquer que le mot-clé `AS`, déjà utilisé pour donner un alias à un attribut, s'utilise aussi pour renommer une table.

```sql
SELECT C.nom_client, C.ville, B.telephone, B.ville
FROM bureau AS B JOIN employe AS E ON E.code_bureau = B.code
JOIN client AS C ON C.res_ventes = E.num_employe
```

Il est important de comprendre que l'opération de jointure crée momentanément une nouvelle table sur laquelle s'effectue la requête `SELECT`. On peut donc combiner les jointures et les opérations de projection, de sélection, de tri, ... vues dans les parties I et II. À titre d'illustration, expliquer ce que produit la requête suivante.

```sql
SELECT nom_client, date_paiement, montant
FROM client JOIN paiement ON client.num_client = paiement.num_client
WHERE montant > 20000
ORDER BY date_paiement, montant
```

Parmi toutes les ventes d'avions, on souhaite savoir lesquelles sont les plus rémunératrices. Pour chaque ligne de bon de commande correspondant à une vente d'avion, afficher le nom de cet avion et le bénéfice total réalisé lors de la vente, compte tenu du prix de vente, du prix d'achat et du nombre d'exemplaires vendus. Afficher seulement les 50 ventes de plus forts bénéfices.

III.3 Autojointure

Dans la table `employe` (voir figure 1), l'attribut `superviseur` est une clé étrangère vers *cette même* table `employe`. Pour reprendre le vocabulaire du chapitre précédent, la table référençante (celle qui possède la clé étrangère) est confondue avec la table cible. Rien n'interdit d'appliquer à ce genre de situation le principe de jointure par clés étudié dans les paragraphes précédents : on parle dans ce cas *d'autojointure*, c'est à dire de jointure d'une table avec elle-même. L'expression d'une autojointure en SQL passe par un renommage de table comme nous allons le voir au travers d'un exemple.

Pour afficher le nom, le prénom et le numéro de superviseur de chaque employé, il suffit d'exécuter une projection.

```sql
SELECT prenom, nom, superviseur FROM employe
```

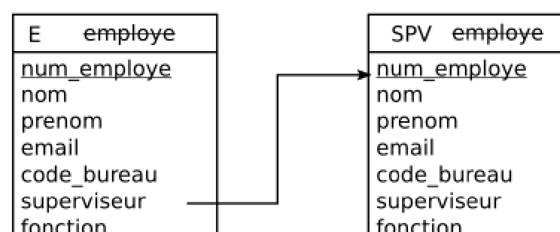
nom	prenom	superviseur
Patterson	Mary	1002
Firrelli	Jeff	1002
Patterson	William	1056
Kato	Yoshimi	1621
Gerard	Martin	1102

Nous voyons que Gerard Martin est supervisé par l'employé 1102, et une recherche visuelle dans la table `employe` nous apprend qu'il s'agit du dénommé Bondur. Plutôt qu'un numéro, nous voulons maintenant afficher directement, à côté du nom et du prénom de chaque employé, *le nom et le prénom de son superviseur*. La requête naïve

```sql
SELECT prenom, nom, superviseur, nom, prenom FROM employe --- idiot !
```

ne produit pas le résultat voulu : elle affiche deux fois le nom et le prénom de chaque employé et mais pas celui de son superviseur !

Tout serait plus simple si nous disposions de deux tables identiques à `employe`, comme sur le schéma ci-dessous. Pour éviter de les confondre, appelons `E` la première et `SPV` la seconde.



Dans ce cas, une jointure entre les deux tables conduirait au résultat souhaité.

```
SELECT E.nom, E.prenom, SPV.nom, SPV.prenom -- si on disposait des deus tables
FROM  E JOIN  SPV
ON E.superviseur = SPV.num_employe
```

En pratique, nous n'allons pas créer de copies permanentes de la table `employe`, mais travailler avec deux alias de cette table, qu'on obtient par renommage et qui n'existeront que durant l'exécution de la requête. Cette démarche astucieuse est à connaître.

```
SELECT E.nom, E.prenom, SPV.nom AS "nom superviseur", SPV.prenom AS "prenom superviseur"
FROM  employe AS E JOIN employe AS SPV -- Deux alias obtenus par renommage
ON E.superviseur = SPV.num_employe ;
```

On parvient ainsi au résultat souhaité (tableau ci-dessous). Pour une meilleur lisibilité du résultat, on a aussi renommé les deux colonnes de droite pour indiquer qu'elles concernent les superviseurs.

nom	prenom	nom superviseur	prenom superviseur
Patterson	Mary	Murphy	Diane
Firrelli	Jeff	Murphy	Diane
Patterson	William	Patterson	Mary
Kato	Yoshimi	Nishi	Mami
Gerard	Martin	Bondur	Gerard

Passons à quelques exercices d'application.

1. Afficher, à côté du nom et du prénom de chaque employé, le numéro de téléphone du bureau dans lequel son superviseur travaille.

2. Afficher, à côté du nom et du prénom de chaque employé, le numéro de téléphone du bureau dans lequel il travaille et le numéro de téléphone du numéro dans lequel son superviseur travaille.

3. Afficher, à côté du nom et du prénom de chaque employé, ceux du superviseur de son superviseur.

IV Fonctions d'agrégation

IV.1 Cinq fonctions à connaître

À l'instar d'un logiciel tableur comme Excell, un SGBD peut calculer des fonctions statistiques en utilisant, pour une colonne donnée, les valeurs présentes sur un ensemble de lignes. Ce genre de calcul s'effectue avec les *fonctions d'agrégation* `MIN`, `MAX`, `SUM`, `AVG` (calcul de moyenne) et `COUNT` (dénombrement des lignes). Voyons quelques exemples élémentaires.

```
SELECT MIN(montant) FROM paiement ;
SELECT MAX(montant) FROM paiement ;
SELECT AVG(montant) FROM paiement ;
SELECT MIN(echelle) FROM article  ;  -- ordre lexicographique sur les chaînes
SELECT MAX(date_commande) FROM commande ; -- idem, commande la plus tardive
SELECT MAX(montant), num_client FROM paiement ; -- *
```

```sql
SELECT SUM(quantite) FROM detail_commande ; -- nb de pièces commandées
SELECT SUM(stock) FROM article ; -- nb total d'articles en stock
```

L'exemple marqué d'une étoile montre qu'on peut aussi obtenir, en plus du minimum, un attribut de la ligne pour laquelle ce minimum est atteint.
Ces fonctions d'agrégation peuvent aussi porter sur des valeurs par des opérations arithmétiques portant sur des attributs. Enfin, l'utilisation de la clause `WHERE` permet de limiter le calcul à certaines lignes. Analyser les trois requêtes ci-dessous pour comprendre ce qu'elles produisent.

```sql
SELECT MAX(prix_unitaire * quantite) FROM detail_commande ;
SELECT AVG(montant) FROM paiement WHERE num_client = 114 ;
SELECT MAX(prix_vente_recommandé - prix_achat) AS "Bénéfice max", nom_article
        FROM article WHERE echelle = "1:10" ;
```

L'opérateur `COUNT` peut s'utiliser sous deux formes.
- `COUNT(*)` compte les lignes ;
- `COUNT(attribut)` compte les lignes où `attribut` n'est pas `NULL`.

Comme notre programme exclut d'utiliser la variable `NULL`[4], les deux formes sont pour nous équivalentes.

```sql
SELECT COUNT(*) FROM employe
SELECT COUNT(nom) FROM employe
```

Attention : lorsqu'un attribut prend la même valeur sur plusieurs lignes, `COUNT(attribut)`, comptabilise chacune d'elles ! Pour compter une seule fois les lignes pour lesquelles cet attribut prend une même valeur, on utilise le mot-clé `DISTINCT`, placé entre les parenthèses suivant `COUNT`.

```sql
SELECT COUNT(*) FROM client ;
-- même résultat ligne suivante : les pays répétés sont comptés plusieurs fois !
SELECT COUNT(pays) FROM client ;
SELECT COUNT(DISTINCT pays) FROM client ; -- Chaque pays est compté une seule fois.
```

1. Dans combien de pays la firme possède-t-elle des bureaux ?

2. Calculer la valeur d'achat de la totalité des articles en stock.

3. Dans combien de lignes de commande l'article `S18_1749` apparaît-il ?

4. Quand l'article `S18_1749` est commandé, en combien d'exemplaires l'est-il en moyenne ?

5. On s'intéresse au client numéro 124. Combien de commandes a-t-il passées ? Combien de lignes de commandes cela représente-t-il (une même commande peut comporter plusieurs lignes) ? En moyenne, combien d'articles identiques commande-t-il à la fois ?

4. Il s'agit d'un mot-clé indiquant les valeurs inconnues.

IV.2 Statistiques sur des groupes

Dans la table `paiement`, chaque client apparaît plusieurs fois et il s'avère confortable, pour une observation visuelle, de regrouper les lignes correspondant à un même client. En affichant seulement le numéro de client et le montant de chaque versement, on obtient ainsi les deux premières colonnes du tableau ci-contre.

Dès lors que les lignes sont regroupées par `num_client`, il est facile de calculer pour chaque groupe quelques paramètres statistiques, tels que le nombre de ligne ou leur montant total. Pour effectuer automatiquement ce traitement, on utilise le mot-clé `GROUP BY`. Voici deux exemples.

num_client	montant	statistiques par groupe
103	6066.78	3 lignes pour un total de 22314,36
103	14571.44	
103	1676.14	
112	14191.12	3 lignes pour un total de 80180.98
112	32641.98	
112	33347.88	
114	45864.03	4 lignes pour un total de 180585.07
114	82261.22	
114	7565.08	
114	44894.74	

```
SELECT num_client,  SUM(montant) FROM paiement GROUP BY num_client ;
--
SELECT num_client, COUNT(*) AS "nb de lignes", AVG(montant), MIN(montant),
   MAX(montant)
FROM paiement GROUP BY num_client  ;
```

num_client	nb de lignes	AVG(montant)	MIN(montant)	MAX(montant)
103	3	7438.12	1676.14	14571.44
112	3	26726.9933333333	14191.12	33347.88
114	4	45146.2675	7565.08	82261.22
119	3	38983.2266666667	19501.82	49523.67

En revanche, la requête suivante est insensée puisqu'il n'existe pas de date commune à toutes les commandes d'un même groupe. Dans ce cas, SQLite ne renvoie pas de message d'erreur, mais un résultat imprévisible.

```
SELECT num_client, date_paiement FROM paiement GROUP BY num_client ; --- insensé !
```

Pour filtrer les réponses en appliquant un critère sur le résultat d'une fonction statistique de sélection, on utilise `HAVING`. Par exemple, pour trouver les numéros et les montants totaux des clients ayant procédé à plus de 100 000€ de paiements, on écrit

```
SELECT num_client, SUM(montant) AS total
FROM paiement GROUP BY num_client HAVING(total > 100000)
```

dans lequel l'utilisation de l'alias `total` éviter de réécrire `SUM(montant)` dans la parenthèse qui suit `HAVING`. Le critère de filtrage peut résulter d'un calcul et ne figure pas obligatoirement dans la liste des attributs de `SELECT`.

```
SELECT num_client, AVG(montant)
FROM paiement GROUP BY num_client HAVING COUNT(*) >= 5
```

Attention à ne pas confondre `WHERE` *et* `HAVING` *!*
La clause `WHERE` sert à sélectionner les lignes d'une table avant tout regroupement, alors que la clause `HAVING` fait porter une condition *sur le résultat d'une fonction d'agrégation* mettant en jeu un groupe. Syntaxiquement, le `WHERE` vient peu après `FROM`, alors que `HAVING` vient après `GROUP BY`. Expliquer ce que font les deux requêtes suivantes en soulignant bien leurs différences.

```
SELECT pays, AVG(limite_de_credit) AS credit_moy
FROM client GROUP BY pays  HAVING credit_moy > 80000 ;

SELECT pays, AVG(limite_de_credit) AS credit_moy
FROM client WHERE limite_de_credit > 80000  GROUP BY pays ;
```

Il est permis d'utiliser de conserve `WHERE` et `HAVING`, à condition de bien distinguer leur signification respective.

```
SELECT res_ventes, MAX(limite_de_credit) AS credit_max
FROM client  WHERE pays = 'USA'
GROUP BY res_ventes HAVING(credit_max > 100000)  ;
```

Cette requête élimine d'abord les clients d'autres pays que les USA. Puis, pour chaque responsable des ventes, on cherche, parmi tous les clients (des USA) dont il a la charge, la plus haute limite de crédit. On l'affiche si elle dépasse 100000 euros.

1. Combien chaque responsable de ventes a-t-il de clients en charge dans chaque pays où il en a au moins un ? Il s'agit d'un exemple de regroupement selon deux critères.

2. Quel est le bénéfice moyen attendu pour chaque catégorie d'articles (motos, trains, avions, voitures, etc) ?

3. Quel est, pour chaque catégorie d'article (avions, trains, etc), le nombre total d'objets en stock et leur valeur d'achat totale.

4. Pour chaque échelle pour laquelle il existe au moins 5 articles distincts, trouver le prix moyen de vente recommandé. Afficher ce prix à côté du nombre d'articles et de l'échelle en triant les lignes du prix moyen le plus bas au prix moyen le plus élevé.

IV.3 Regroupements et jointures

Le regroupement de lignes peut s'utiliser suite à une jointure, ce qui donne lieu à des requêtes relativement complexes. Pour bien comprendre exemples qui suivent, il est commode de considérer que le SGBD effectue d'abord la jointure et d'imaginer la table qui en résulte. Puis on envisage le regroupement, et enfin on opère le traitement statistique de chacun de ces groupes [5]. Expliquons le rôle de chacune des requêtes suivantes.

```
1 SELECT nom_client, SUM(montant)
2 FROM client JOIN paiement ON client.num_client = paiement.num_client
  GROUP BY nom_client ;
```

```
1 SELECT num_client, SUM(quantite) AS 'Nb d''objets', SUM(quantite * prix_unitaire) AS
     'prix total'
2 FROM commande AS C JOIN detail_commande AS D ON C.num_commande = D.num_commande
  GROUP BY num_client ;
```

```
1 SELECT pays, COUNT(*) AS 'Nb de lignes', SUM(quantite) AS 'Nb d''objets',
     SUM(quantite*prix_unitaire) AS 'CA'
```

5. Pendant le cours, le professeur affiche dans un premier temps le résultat de la jointure et y fait apparaître les groupes, puis dans un second temps il effectue les requêtes complètes.

```
FROM client  JOIN commande AS C ON client.num_client = C.num_client
          JOIN detail_commande AS D ON C.num_commande = D.num_commande
GROUP BY pays;
```

Écrire maintenant les requêtes répondant aux questions suivantes.

1. Trouver le bénéfice total généré par les articles en provenance de chaque fournisseur. Les ordonner du plus grand au plus petit.

2. Afficher le nom et le prénom de chaque superviseur, avec le nombre de ses subordonnés.

3. La direction est contrariée par le fait que certains clients multiplient les petits versements ; elle préfère au contraire les versements de montant élevé. Elle décide d'organiser un concours pour récompenser les trois bureaux dont les clients procèdent, en moyenne, à des versements les plus élevés. Pour que la moyenne soit significative, seuls les bureaux ayant reçu au moins 30 versements participent à la compétition. Écrire la requête qui renvoie le code et la ville des trois bureaux à récompenser, accompagnés du versement moyen dans chacun.

4. Reprendre la question précédente en enlevant la contrainte des 30 versements, en classant *tous* les bureaux (et pas seulement les trois premers), et afficher le nombre de clients dont chaque bureau a reçu des versements.

V Des requêtes dans les requêtes

Le résultat d'une requête est soit une table (cas le plus fréquent), soit une valeur unique (avec une fonction d'agrégation par exemple). Ce résultat peut lui-même être utilisé à l'intérieur d'une autre requête, que ce soit dans la table source mentionnée après `FROM`, dans les conditions à vérifier dans `WHERE` et `HAVING`, ou même dans la liste d'attributs qui suit immédiatement `SELECT`. Nous allons voir quelques exemples de requêtes utilisant cette possibilité. Pour les comprendre, il est commode d'analyser d'abord ce que produit la requête interne, qu'on appelle aussi sous-requête.

Voici un premier exemple avec une sous-requête après `FROM` ; le résultat intermédiaire constitue une source de ligne pour la requête principale. Noter la nécessité de placer la sous-requête entre parenthèses.

```
1 SELECT num_client , MAX(total) FROM
  (SELECT num_client , SUM(montant) AS total FROM paiement GROUP BY num_client )
```

Voici un second exemple avec une sous-requête dans la liste de données à extraire qui suit `SELECT`. C'est ici possible parce que la sous-requête renvoie une valeur unique pour chaque ligne du résultat de la requête principale.

```
1 SELECT code_article , prix_vente_recommandé ,
  (SELECT AVG(prix_unitaire) FROM detail_commande WHERE detail_commande.code_produit =
     code_article) AS 'prix de vente moyen'
  FROM article;
```

Cet exemple n'est pas très pertinent : on obtient plus naturellement le même résultat avec une jointure et un regroupement.

Voyons enfin une situation dans laquelle la sous-requête est utilisée par la clause `WHERE`.

```
1 SELECT nom_client , limite_de_credit
  FROM client
  WHERE limite_de_credit > (SELECT AVG(limite_de_credit) FROM client)
```

Écrire les requêtes répondant aux questions suivantes.

1. Trouver la moyenne du nombre de subordonnés que possèdent les employés qui en ont au moins un.

2. Pour chaque pays possédant au moins 2 clients, écrire le nom de ce pays, le nombre de clients qui y résident, le numéro du client de ce pays qui produit le plus de chiffre d'affaire, et la valeur de ce chiffre d'affaire (question trop difficile !).

VI Conseils pratiques et aide-mémoire SQL

Contrairement à ce que vous faites couramment en Python, on n'écrira pas de long code SQL, mais des requêtes constituées de quelques lignes seulement. En vue des TD, voici quelques conseils pratiques pour leur rédaction.

- Les mots-clés du langage SQL peuvent indifféremment être écrits en majuscule ou en minuscule. Pour améliorer la lisibilité des codes, je vous demande de les écrire en majuscule.
- Contrairement à Python, SQL n'accorde aucune signification syntaxique aux retours à la ligne et aux indentations. Pour plus de lisibilité, présenter les requêtes complexes sur plusieurs lignes.
- Pour écrire un commentaire, on le fait précéder de deux tirets.
- Pour faciliter la lecture des codes, n'hésitez pas à préfixer le nom des attributs par celui de la table dont ils sont extraits, en écrivant par exemple `client.pays` plutôt que `pays`. L'éditeur que nous utiliserons possède un système d'autocomplétion : si vous tapez `client.` (le nom de la table suivi d'un point), la liste des attributs apparaît et il vous suffit de sélectionner celui dont vous avez besoin.
- Pour séparer les unes des autres les différentes requêtes SQL placées dans un même fichier, on termine chacune d'elles par un point-virgule.
- Lorsqu'un nom de variable possède une espace, on l'écrit entre doubles apostrophes, `"comme ceci"`.
- L'ordre des mots-clés est rappelé sur l'aide-mémoire ci-dessous.

```
1 SELECT *
  FROM table -- la table peut être issue de jointures
  WHERE condition
  GROUP BY expression
5 HAVING condition
  { UNION | INTERSECT | EXCEPT }
  ORDER BY expression
  LIMIT count
  OFFSET start
```

ANNEXE : BDDR `modeles_reduits`

Une entreprise commercialise des modèles réduits de voitures, d'avions, de trains, etc. Ses clients, des magasins de jouets situés dans le monde entier, lui passent des commandes et procèdent à des paiements par virement bancaire (sans lien direct avec les commandes). Cette entreprise possède plusieurs dizaines d'employés, chacun rattachés à l'un de ses bureaux régionaux. La BDD correspondante est formée de 7 tables.

- table `article(code_article : entier, nom_article : chaîne, categorie : chaîne, echelle : chaîne, fournisseur : chaîne, prix_achat : flottant, prix_recommandé : flottant)`
  Des exemples de catégories possibles sont `Classic Cars, Trucks, Trains`.
- table `client(num_client : entier, nom : chaîne, pays : chaîne, ville : chaîne, telephone : chaîne, adresse : chaîne, code_postal : chaîne, resp_ventes : entier, credit_limite : flottant)`. L'attribut `resp_ventes` permet de savoir quel employé est chargé de ce client.
- table `employe(num_employe : entier, nom : chaîne, prenom : chaîne, email : chaîne, code_bureau : entier, superviseur : entier, fonction : chaîne)`
  L'attribut `code_bureau` indique dans quel bureau chaque employé travaille et l'attribut `superviseur` permet de connaître son supérieur hiérarchique.
- table `bureau(code : entier, pays : chaîne, ville : chaîne, telephone : chaîne, adresse : chaîne, code_postal : chaîne)`
- table `commande(num_commande : entier, date_commande : chaîne, date_requise : chaîne, date_livraison : chaîne, statut : chaîne, num_client)`
  L'attribut `statut` permet de savoir si la commande a été livrée (`shipped`), annulée (`canceled`) ou autre.
- table `detail_commande(num_commande : entier, code_produit : entier, quantité : entier, prix_unitaire : entier, ligne_bon_commande : entier)`
  Sa clé primaire est formée de deux attributs ; pour une commande et un produit donné, on indique la quantité et le prix unitaire. Cette table établit intervient dans deux associations un-à-plusieurs pour décomposer l'association plusieurs-à-plusieurs qui lierait directement le type-entité `article` et le type-entité `commande` (un même article peut figure dans plusieurs commandes, et un un même commande peut comporter plusieurs articles).
- table `paiement(num_client : entier, num_virement : entier, date : chaîne, montant : flottant)`

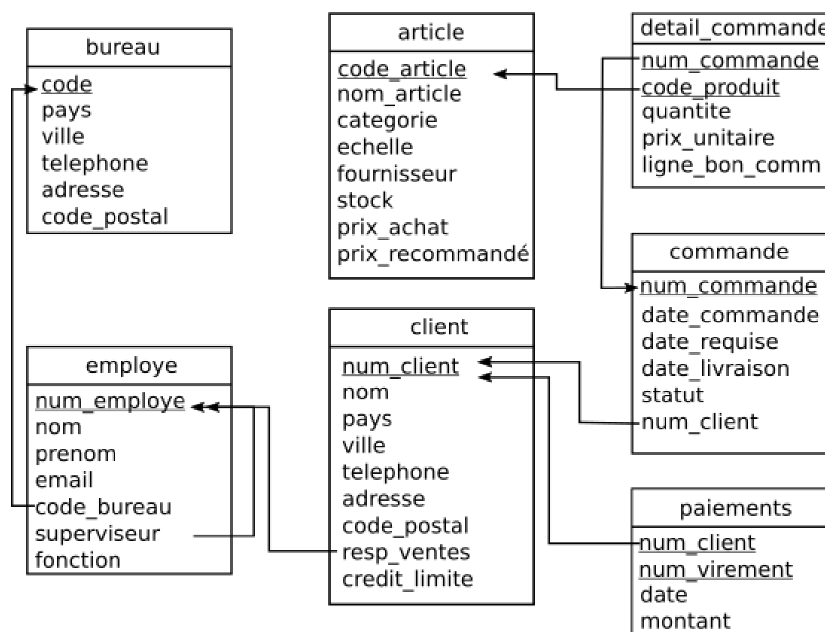


FIGURE 1 – Base de données `modeles_reduits`