SCI6306 Bases de données documentaires (Automne 2023)

Christine Dufour, EBSI, UdeM

A2023 22 septembre 2023

Cours 3 : SQL (partie 1 de 3)

SCI6306

Christine Dufour, EBSI, UdeM

Table des matières

Cours 3 - SQL (partie 1 de 3)

- + Au programme aujourd'hui
- Environnements de pratique SQL
- SQL (niveau 1)
    - Requête Sélection
        - Requête Sélection : SELECT restreint
        - Requête Sélection : valeur NULL et chaîne vide
        - Requête Sélection : Jointures
            - Équijointure à deux tables
            - Équijointure à trois tables
    - Principe d'indépendance
- Travail individuel sur les requêtes SQL
- Ressources en lien avec le cours

1. + Au programme aujourd'hui

- Tour rapide des **environnements de pratique**
- **SQL – partie 1 de 3** (*niveau 1*)
    - SELECT restreint
    - Alias
    - Conditions
    - Tri
    - Expressions
    - NULL vs chaîne vide
    - SELECT avec jointures
- Laboratoire : **TP Requêtes SQL** – niveau 1

2. Environnements de pratique SQL

Vous pourrez exploiter deux environnements de pratique pour SQL, décrits ci-dessous : (1) directement dans *phpMyAdmin*, et (2) l'environnement *KeSQ^{i}L fait?*

1. **Pratique sur phpMyAdmin** (https://www.gin-ebsi.umontreal.ca/phpmyadmin/) :
    - Connectez-vous en utilisant les **informations de connexion pour votre compte individuel** envoyées par courriel
    - Vous trouverez de l'aide sur l'interface de phpMyAdmin à https://cours.ebsi.umontreal.ca/sci6306/co/site_phpmyadmin.html
2. **Environnement KeSQ^{i}L fait?** (https://dufour.ebsi.umontreal.ca/mooc_bd/)

- Cet environnement propose des **exercices SQL avec solutionnaires** ainsi qu'un **espace de pratique libre** sur deux bases de données dont INSCRIP. Les exercices sont classés entre autres par niveau de difficultés, le niveau 1 correspondant à la matière vue au premier cours sur SQL, le niveau 2 au deuxième cours et le niveau 3 au troisième cours. Seules les requêtes de type "Sélection" sont couvertes (*select*).

3. SQL (niveau 1)

SQL est une **norme ISO/IEC** qui précise la syntaxe et les règles pour effectuer **toutes les interactions** avec une base de données relationnelle (interrogation, mise à jour, gestion).

On y retrouve deux types de langage :

- **Data Definition Language** (DDL) : langage qui permet de créer les différents objets (tables de données, index, etc.), p. ex., CREATE TABLE
- **Data Manipulation Language** (DML) : langage qui permet de manipuler les données comme, p. ex., des requêtes Sélection (SELECT) pour extraire des données, des requêtes Ajout (INSERT INTO) pour ajouter des données, des requêtes Mise à jour (UPDATE) pour mettre à jour des données ou des requêtes Suppression (DELETE) pour effacer des données.

Nous nous attarderons dans le cadre du cours uniquement au *DML*, phpMyAdmin (ou d'autres interfaces visuelles permettant de gérer les interactions avec MySQL) prenant en charge les requêtes relevant du *DDL*. Les grandes lignes de ce langage seront présentées dans les trois cours dédiés à SQL et illustrées à partir de la BD INSCRIP. Le travail individuel sur les requêtes SQL permettra de mettre les notions abordées en classe en pratique.

3.1. Requête Sélection

Les **requêtes Sélection** sont les requêtes les plus fréquemment utilisées avec une base de données relationnelle afin de **repérer** des données correspondant à certains critères, les **extraire** d'une ou plusieurs tables et les **afficher** sous forme d'un tableau (les colonnes représentant les champs que l'on veut afficher et les lignes, les valeurs des champs pour chacun des enregistrements correspondant à la requête. La forme générique d'une requête *Sélection* est la suivante :

```
1 SELECT nom(s) de champ et/ou valeur(s) fixe(s)
2 FROM nom(s) de table
3 WHERE condition(s)
4 GROUP BY nom(s) de champ pour regrouper
5 ORDER BY nom(s) de champ pour le tri;
```

Les éléments en minuscule sont ceux à personnaliser en fonction du besoin d'information. Les différents "mots réservés" (en majuscules dans l'exemple, les majuscules n'étant toutefois pas obligatoires) utilisés dans une requête SQL sont en soi assez parlants :

- *Ligne 1* : Le tout premier mot indique le **type de requête**. *SELECT* précise qu'il s'agit d'une requête pour extraire des données. Ce qui suit ce premier mot se trouve être les différentes colonnes qui seront affichées dans la table de données retournée par la requête. Cette première ligne est obligatoire.
- *Ligne 2* : La clause **FROM** précise la source de données dans laquelle la requête sera exécutée. Ce peut être une seule table, mais aussi plusieurs. Comme nous le verrons plus tard, ce pourrait même être une requête! Cette clause est aussi obligatoire.
- *Ligne 3* : La clause **WHERE** indique les critères particuliers afin de retenir un enregistrement. Elle est facultative comme il est possible qu'il n'y ait aucune restriction particulière. Si plusieurs critères sont appliqués, ils sont séparés par l'opérateur booléen approprié par exemple *AND* si on veut que tous les critères soient appliqués.
- *Ligne 4* : La clause **GROUP BY** sert à indiquer un ou des champs qui serviront à regrouper les données extraites par exemple pour faire des calculs. Tout comme la clause *WHERE*, elle est facultative comme tous les besoins d'information ne nécessitent pas de regroupements.

- *Ligne 5* : La clause **ORDER BY** vient indiquer le ou les champs qui serviront pour trier les résultats de la table de données retournée par la requête. Elle aussi est facultative; en son absence, ce sera l'ordre de tri par défaut qui sera appliqué.

Une synthèse de la forme générique d'une requête **Sélection** est présente dans la Documentation complémentaire du cours[1].

a) Requête Sélection : SELECT restreint

Un *SELECT restreint*∗ est une requête SQL dans laquelle une seule table de données est utilisée comme source des données dans la clause *FROM*. Différents exemples exploitant la BD INSCRIP seront présentés pour en découvrir ses particularités. Pour chacun des exemples, le besoin d'information sera précisé ainsi que décomposé en ses différentes parties (**cette décomposition est très importante pour s'assurer de bien comprendre le besoin et le traduire efficacement en SQL**). Finalement la requête SQL correspondante sera présentée ainsi que la table de données résultante.

Exemple **: Besoin d'information : liste de toutes les informations "descriptives" sur les étudiant.e.s**

Décomposition du besoin :

- *Information affichée* : Tous les champs décrivant les étudiant.e.s
- *Source de données* : Table ETUD

Requête SQL :

```
1 SELECT *
2 FROM etud;
```

no_etud	nom	adresse	dat_nais	prog
10001	Wagner, Richard	1200 de l'Opéra, Bayreuth	1980-01-01	Musique
10002	Bretécher, Claire	2400 du Fou-rire, Paris	1950-01-01	Littérature
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	Sciences humaines
10004	Asimov, Isaac	9600 du Futur, Los Angeles		Sciences
10005	Simenon, Georges	9700 des Lilas, Montréal	1960-01-01	Littérature
10006	Smithwick, Dale	4300 Crescent, Montréal		Sciences

Table des résultats

À noter : l'utilisation de l'**astérisque** après le mot-clé *SELECT* remplace la précision fine des champs à afficher. Tous les champs de la table de données source seront en ce cas affichés. Il ne faut toutefois pas abuser de ce truc car, lorsque des tables comportent plusieurs champs, la table résultante peut se trouver alourdie peut-être sans nécessité (certains champs ne sont peut-être pas pertinents au besoin d'information).

Exemple **: Besoin d'information : liste des noms des étudiant.e.s et de leur numéro d'étudiant.e**

Décomposition du besoin :

- *Information affichée* : Noms des étudiant.e.s et numéros des étudiant.e.s
- *Source de données* : Table ETUD

Requête SQL :

```
1 SELECT nom, no_etud
2 FROM etud;
```

[1] https://cours.ebsi.umontreal.ca/sci6306/co/structure_requete_selection.html

nom	no_etud
Wagner, Richard	10001
Bretécher, Claire	10002
Tremblay, Michel	10003
Asimov, Isaac	10004
Simenon, Georges	10005
Smithwick, Dale	10006

Table des résultats

À noter : on retrouve, suite au *SELECT*, les noms des deux champs qui nous intéressent séparés par une virgule. La syntaxe complète pour indiquer des noms de champs est *nom de la table*.*nom du champ*. Si on peut ici se passer d'indiquer le nom de la table, c'est du fait de n'avoir qu'une seule table dans la clause *FROM*. Il n'y a ainsi **aucune ambiguïté possible** quand à la table d'où ces champs proviennent.

Lorsque l'on fait des requêtes avec plus d'une table, si le nom d'un champ n'existe que dans une seule table, il n'est pas nécessaire d'indiquer le nom de la table. Cependant, si deux tables possèdent un champ portant le même nom, il est nécessaire d'indiquer le nom de la table pour désambiguïser la source. Si on fait un parallèle avec des noms de personne, si dans un groupe il n'y a qu'un seul Paul, il n'est pas nécessaire de préciser son nom de famille pour l'identifier. S'il y avait deux personnes se prénommant Paul, le nom de famille devient nécessaire!

Exemple : Besoin d'information : Liste des noms des étudiant.e.s présent.e.s dans la table ETUD, de leur numéro d'étudiant.e et de leur âge

Décomposition du besoin :

- *Information affichée* : Noms des étudiant.e.s, numéros des étudiant.e.s, **âge des étudiant.e.s**
- *Source de données* : Table ETUD

Requête SQL :

```
1 SELECT nom, no_etud, round(datediff(now(),dat_nais)/365,0) as Âge
2 FROM etud;
```

nom	no_etud	Âge
Wagner, Richard	10001	38
Bretécher, Claire	10002	68
Tremblay, Michel	10003	48
Asimov, Isaac	10004	
Simenon, Georges	10005	58
Smithwick, Dale	10006	

Table des résultats

À noter : Comme l'âge ne fait pas partie des données directement comprises dans la table ETUD, il faut le *déduire* à partir de la date de naissance. Heureusement pour nous, SQL comprend plusieurs **fonctions textuelles**, **mathématiques**, **statistiques**, etc. Comment déduire l'âge à partir d'une date de naissance? En faisant la différence entre la date actuelle et la date de naissance! La fonction **datediff()** permet de calculer le nombre de jours entre deux dates. Cette fonction comporte deux *paramètres*∗ que l'on doit indiquer entre parenthèses à la suite de son nom, et séparés par une virgule. Le premier paramètre est la date la plus récente, le deuxième paramètre est la date la plus ancienne. On peut indiquer en paramètre directement une date, comme on peut faire référence à un nom de champ (p. ex. ici *dat_nais* en deuxième position), voire même à une autre fonction. C'est ce que l'on fait avec la fonction **now()** qui permet d'aller chercher la date du jour.

Comme la fonction **datediff** retourne un nombre de jours, pour approximer le nombre d'années, il faut diviser le nombre retourné par 365. Finalement, si l'on veut avoir un nombre d'années entier, sans décimale, il est possible d'exploiter la fonction **round()** qui possède deux paramètres soit (1) le nombre à arrondir (ici la fonction **datediff()**) et (2) le nombre de décimales désiré (ici 0).

Un dernier élément à remarquer est l'utilisation d'un **alias** suite au calcul de l'âge (`as Âge`). Cet alias n'est pas obligatoire; il s'agit en fait de ce qui apparaîtra comme en-tête de la colonne pour les âges. Si vous n'indiquez rien, vous retrouverez comme en-tête de cette colonne la fonction au complet (`round(...)`) ce qui n'est probablement pas très compréhensible pour un.e utilisateur.trice externe. Les alias servent donc à rendre les tables de données **plus lisibles**.

Exemple : Besoin d'information : Liste des informations relatives aux étudiant.e.s né.e.s après le 1er janvier 1960

Décomposition du besoin :

- *Information affichée* : Tous les champs décrivant les étudiant.e.s
- *Source de données* : Table ETUD
- ***Condition*** : être né.e après le 1er janvier 1960

Requête SQL :

```sql
SELECT *
FROM etud
WHERE dat_nais > '1960-01-01';
```

no_etud	nom	adresse	dat_nais	prog
10001	Wagner, Richard	1200 de l'Opéra, Bayreuth	1980-01-01	Musique
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	Sciences humaines

Table des résultats

À noter : la clause *WHERE* retourne toujours soit **VRAI**, soit **FAUX**. Il faut donc la construire en ce sens. Pour valider si quelqu'un est né après le 1er janvier 1960, il faut exploiter l'**opérateur de relation "plus grand" (>)** pour comparer la date de naissance d'un.e étudiant.e (champ DAT_NAIS) au premier janvier 1960. Si la condition retourne VRAI, la ligne de la table de données source sur laquelle la condition a été vérifiée sera présente dans la table des résultats. Si la condition est fausse pour une ligne, cette ligne ne se trouvera pas dans la table des résultats.

Il est important de comprendre qu'une requête SQL travaille de manière "**linéaire**" en traitant chacune des lignes de la (ou des) table(s) de la clause *FROM* à la fois, indépendamment les unes des autres. Ici, la requête SQL regarde chacune des lignes de la table ETUD et vérifie si le contenu du champ DAT_NAIS est plus grand ou non à la date du 1er janvier 1960. Si oui, comme c'est le cas pour M. Wagner, cette ligne se retrouvera dans la table des résultats. Si non, comme c'est le cas pour Mme Bretécher, cette ligne ne sera pas dans la table résultante.

Il est aussi à noter que les dates indiquées explicitement dans une requête SQL dans MySQL - la syntaxe peut être différente dans d'autres SGBD relationnels - sont présentées entre **guillemets simples** (') ou **guillemets doubles** ("). Les deux fonctionneront. Vous remarquez que les **guillemets simples sont privilégiés** dans le matériel du cours; vous comprendrez pourquoi un peu plus tard dans la session lorsque nous commencerons l'intégration d'une base de données avec PHP.

Exemple : Besoin d'information : Liste des informations relatives aux étudiant.e.s né.e.s après le 1er janvier 1960 et habitant à Montréal

Décomposition du besoin :

- *Information affichée* : Tous les champs décrivant les étudiant.e.s
- *Source de données* : Table ETUD
- ***Conditions*** : être né.e après le 1er janvier 1960 **et habiter Montréal**

Requête SQL :

```sql
SELECT *
FROM etud
WHERE dat_nais > '1960-01-01' AND adresse like '%Montréal%';
```

no_etud	nom	adresse	dat_nais	prog
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	Sciences humaines

Table des résultats

À noter : cet exemple illustre la possibilité d'utiliser des **opérateurs booléens** dans une clause *WHERE*, le *AND* permettant par exemple de s'assurer de l'application de plus d'un critère. En sus du *AND*, le *OR* et le *NOT* peuvent aussi être utilisés.

Autre élément à noter, l'utilisation de l'opérateur **like** qui permet de comparer un champ de type caractères à une chaîne de caractères (qui doit être entre guillemets simples ou doubles, les simples étant ici aussi privilégiés). L'opérateur **like** fait à la base une comparaison *exacte*. Si vous écriviez `like 'Montréal'`, cette condition ne serait vraie que pour les étudiant.e.s où on retrouve uniquement "Montréal" dans le champ ADRESSE. Or il s'avère que ce champ contient une adresse plus détaillée que le seul nom de la ville de résidence. Il faut donc utiliser la **troncature**, représentée dans MySQL par le **signe de pourcentage**, devant mais aussi après la chaîne `Montréal` (*%Montréal%*).

¶ *Syntaxe* : Requête Sélection : Expressions (en résumé)

Les **expressions** pouvant être incluses dans une requête peuvent contenir des :

- **Constantes** (nombre, chaîne de caractères entre guillemets simples, date entre guillemets en format complet, true/false)
- **Noms de champs**
- **Noms de variables** (@nom qui permet à l'usager de paramétrer la requête)
- **Opérateurs de transformation** (+, -, *, /)
- **Opérateurs de relation** (=, <, >, <>, LIKE, IN, IS)
- **Noms de fonction** (CONCAT, UCASE, LCASE, DATEDIFF, IF, DATE_FORMAT, etc.)
- **Opérateurs booléens** (AND, OR, NOT)
- **Parenthèses**

Pour plus d'information sur les expressions, voir la Documentation complémentaire du cours[1].

🔎 *Exemple* : Besoin d'information : Liste des informations relatives aux étudiant.e.s né.e.s après le 1er janvier 1960 et habitant à Montréal, de la ou du plus jeune à la plus vieille ou au plus vieux

Décomposition du besoin :

- *Information affichée* : Tous les champs décrivant les étudiant.e.s
- *Source de données* : Table ETUD
- ***Conditions*** : être né.e après le 1er janvier 1960 **et habiter Montréal**
- ***Tri*** : **par ordre croissant d'âge**

Requête SQL :

```sql
SELECT *
FROM etud
WHERE dat_nais > '1960-01-01' AND adresse like '%Montréal%'
ORDER BY dat_nais DESC;
```

[1] https://cours.ebsi.umontreal.ca/sci6306/co/expressions_mysql.html

no_etud	nom	adresse	dat_nais	prog
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	Sciences humaines

Table des résultats

À noter : on peut indiquer plus d'un champ dans une clause *ORDER BY* si plus d'une clé de tri est nécessaire. Si c'est le cas, les différents champs servant de clés de tri sont séparés par une virgule, par exemple `ORDER BY date_nais, nom`. On ajoute une deuxième clé de tri, par exemple, si on sait que pour la première clé de tri une valeur peut apparaître plus d'une fois. Par exemple, pour la date de naissance, il pourrait y avoir plus d'une personne associée. Dans ce cas, y a-t-il un ordre particulier qui pourrait nous intéresser? Si oui, on ajoute ce deuxième champ au tri.

Dans l'exemple ci-dessus, une seule clé a été utilisée, soit la date de naissance, comme il n'y avait aucune préférence particulière lorsque deux personnes possèdent la même date de naissance. Le mot-clé **DESC** qui suit le nom du champ sert à préciser que l'on veut l'ordre décroissant. Si on veut l'ordre croissant, on peut soit indiquer *ASC*, soit ne rien mettre comme c'est l'ordre croissant qui est défini par défaut. Chaque champ servant de clé de tri peut être ordonné indépendamment des autres, par exemple `ORDER BY date_nais ASC, nom DESC` c'est à dire de mettre en ordre croissant de date de naissance (donc de la personne la plus âgée à la plus jeune) et ensuite, au besoin, en ordre alphabétique décroissant de nom.

b) Requête Sélection : valeur NULL et chaîne vide

Dans certains contextes, on peut vouloir distinguer le fait qu'une donnée est absente – p. ex. un oubli à la saisie ou un refus de la donner, du fait que l'information n'existe pas (p. ex.un numéro de cellulaire pour quelqu'un n'ayant pas de téléphone cellulaire). On peut ainsi utiliser la **valeur NULL** pour une donnée absente ou manquante, et la **chaîne vide** pour une donnée qui n'existe pas (pour les champs textuels de type "texte" seulement). *Valeur NULL* et *chaîne vide* ont toutefois des comportements différents en recherche qu'il faut connaître.

Valeur NULL

- Une **valeur NULL n'est pas une chaîne de caractères**; elle retournera ainsi toujours FAUX dans une condition si on utilise les opérateurs de relation <, >, =, <=, >=
- Pour repérer les champs ayant une valeur *NULL*, il faut ainsi plutôt exploiter l'opérateur **is** ainsi *champ* `is NULL`. On peut aussi utiliser la fonction `isnull(`*champ*`)` qui retourne VRAI si le champ contient une valeur NULL et FAUX sinon.
- Si au contraire ce sont les champs ne comportant pas la valeur *NULL* qui nous intéresse, il faut exploiter le booléen *NOT* : *champ* `is not NULL` ou `not isnull(`*champ*`)`

Chaîne vide

- La **chaîne vide est une chaîne de caractères** de longueur 0.
- Pour repérer les champs vides, on peut utiliser l'opérateur de relation égal (=), la chaîne vide étant représentée par deux guillemets simples côte-à-côte (ou deux guillemets doubles) : *champ* `= ''`
- Pour repérer les champs non vides, c'est la juxtaposition des opérateurs de relation plus petit et plus grand, qui signifie "est différent de" qui peut être exploitée : *champ* `<> ''`

Pour des explications complémentaires sur la différence entre la valeur *NULL* et la *chaîne vide*, voir la Documentation complémentaire du cours[1].

[1] https://cours.ebsi.umontreal.ca/sci6306/co/null_vide.html

c) Requête Sélection : Jointures

Le modèle relationnel permet **plusieurs tables** dans une même base de données pour éliminer la redondance et permettre de recréer, au besoin, les occurrences multiples. On peut ainsi vouloir exploiter **plus d'une table** de données dans une requête SQL. En ce cas, il faut faire **une ou des jointures** pour les mettre en relation :

- Les jointures utilisent les **clés externes/étrangères** définies entre les tables.
- La forme de jointure la plus courante est l'***équijointure*** $*$ (c'est-à-dire jointure basée sur l'égalité des valeurs des clés externes).

Voir la Documentation complémentaire du cours[1] pour une explication détaillée de la notion de jointure.

i) Équijointure à deux tables

Besoin d'information : Liste des cours et des professeur.e.s qui les donnent

Imaginons que nous voulions obtenir la liste des cours et des professeur.e.s qui les donnent. Nous aurions besoin en ce cas à la fois de la table COURS et de la table PROF, qu'il nous faudrait "accrocher" l'une par rapport à l'autre sur la base du champ NO_PROF qui est la clé externe dans la table COURS qui est associée à la clé primaire de la table PROF :

Table COURS

no_cours*	titre	local	no_prof
20001	Histoire 101	C-2044	30001
20002	Géographie 302	T-5334	30002
20003	Philosophie 813		30003
20004	Arts plastiques 304	T-5334	
20005	Mathématique 307	C-2044	30003
20006	Chimie 110	H-2034	30002
20007	Éthique 007		

**clé primaire*

clé externe

Table PROF

no_prof*	nom	bureau
30001	Simard, Vianney	H-2034
30002	Turmel, Rémi	T-5334
30003	Asimov, Isaac	A-1000
30004	Valjean, Jean	H-2034

**clé primaire*

ÉQUIJOINTURE

Résultats

no_cours	titre	local	no_prof	nom	bureau
20001	Histoire 101	C-2044	30001	Simard, Vianney	H-2034
20002	Géographie 302	T-5334	30002	Turmel, Rémi	T-5334
20003	Philosophie 813		30003	Asimov, Isaac	A-1000
20005	Mathématique 307	C-2044	30003	Asimov, Isaac	A-1000
20006	Chimie 110	H-2034	30002	Turmel, Rémi	T-5334

Illustration d'une équijointure à deux tables

Cette idée "d'**accrocher**" les tables (donc de les joindre) est **importante**. Si vous ne le faites pas explicitement dans votre requête SQL, le SGBD relationnel ne comprendra pas de lui-même qu'il ne faut retenir pour un cours que les informations qui correspondent au professeur ou à la professeure qui l'enseigne. Sans une jointure explicite, un cours sera associé à tous.tes les professeur.e.s sans exception!

Décomposition du besoin :

- *Information affichée* : Information sur les cours et sur les professeur.e.s qui les enseignent
- *Sources de données* : Tables COURS et PROF
- ***Jointure*** : entre la table COURS et la table PROF sur la base du champ NO_PROF

Requête SQL :

```sql
SELECT no_cours, titre, local, cours.no_prof, nom, bureau
FROM cours, prof
WHERE cours.no_prof=prof.no_prof;
```

[1] https://cours.ebsi.umontreal.ca/sci6306/co/documentation_complementaire_2.html

À noter : Dans la requête ci-dessus, c'est la clause *WHERE* qui permet de s'assurer qu'un cours n'est associé qu'au professeur ou à la professeure qui l'enseigne. C'est une **condition de jointure**.

Une autre syntaxe peut être utilisée en SQL pour représenter une jointure :

```
SELECT no_cours, titre, local, prof.no_prof, nom, bureau
FROM cours INNER JOIN prof ON cours.no_prof = prof.no_prof;
```

Au lieu d'une condition sur la clé externe dans une clause *WHERE*, la jointure est faite directement dans la clause *FROM* via l'opérateur *INNER JOIN ... ON*. Cependant, lorsque l'on travaille directement en SQL, il est plus simple de procéder de la première manière. L'opérateur *INNER JOIN* devient rapidement complexe lorsqu'il y a plus de deux tables à joindre, surtout si on débute.

ii) Équijointure à trois tables

Besoin d'information : Liste des étudiant.e.s et des cours suivis avec la note obtenue

Imaginons que cette fois-ci, ce qui nous intéresse est une liste des étudiant.e.s et des cours suivis avec la note obtenue. Afin d'obtenir toutes les données nécessaires, il nous faudra exploiter les tables ETUD, SUIT et COURS. En ce cas, il ne s'agit pas d'une seule jointure, mais bien de **deux jointures distinctes**, soit une pour "accrocher" les tables SUIT et ETUD et une autre pour "accrocher" les tables SUIT et COURS :

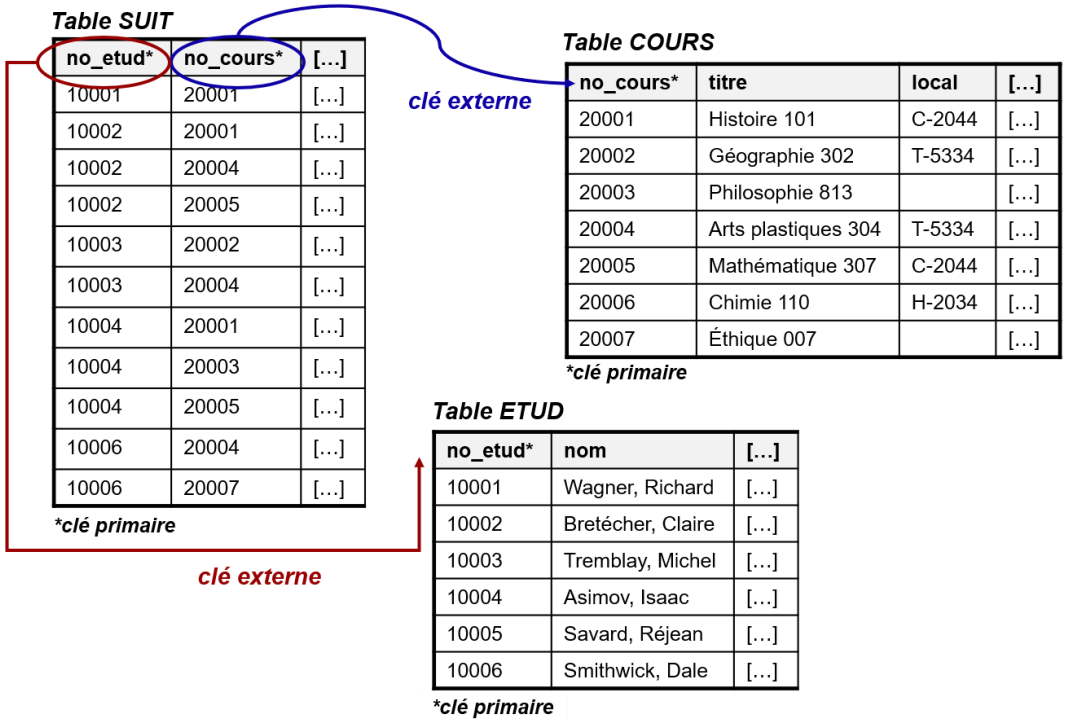

Illustration d'une équijointure à trois tables

Décomposition du besoin :

- *Information affichée* : Information sur les étudiant.e.s et sur les cours suivis avec leur note
- *Sources de données* : Tables ETUD, SUIT et COURS
- ***Jointures*** :
    - entre la table SUIT et la table COURS sur la base du champ NO_COURS
    - entre la table ETUD et la table SUIT sur la base du champ NO_ETUD

Requête SQL :

```
SELECT etud.no_etud, nom, adresse, dat_nais, cours.no_cours, titre, local, no_prof, note,
note_p
FROM cours, etud, suit
WHERE cours.no_cours=suit.no_cours
AND etud.no_etud=suit.no_etud;
```

no_ etud	nom	adresse	dat_ nais	no_ cours	titre	local	no_ prof	note	note _p
10001	Wagner, Richard	1200 de l'Opéra, Bayreuth	1980-01-01	20001	Histoire 101	C-2044	30001	75.50	Oui
10002	Bretécher, Claire	2400 du Fou-rire, Paris	1950-01-01	20001	Histoire 101	C-2044	30001	0.00	Non
10002	Bretécher, Claire	2400 du Fou-rire, Paris	1950-01-01	20004	Arts plastiques 304	T-5334		0.00	Non
10002	Bretécher, Claire	2400 du Fou-rire, Paris	1950-01-01	20005	Mathématique 307	C-2044	30003	98.00	Oui
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	20002	Géographie 302	T-5334	30002	80.00	Oui
10003	Tremblay, Michel	4800 St-Laurent, Montréal	1970-01-01	20004	Arts plastiques 304	T-5334		0.00	Non
10004	Asimov, Isaac	9600 du Futur, Los Angeles		20001	Histoire 101	C-2044	30001	0.00	Oui
10004	Asimov, Isaac	9600 du Futur, Los Angeles		20003	Philosophie 813		30003	38.00	Oui
[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]

Table des résultats

À noter : Les deux jointures se retrouvent dans la clause *WHERE* liées par l'opérateur booléen *AND* comme on veut que les deux jointures soient appliquées.

L'opérateur *INNER JOIN ... ON* pourrait ici aussi être utilisé au lieu d'indiquer les jointures dans la clause *WHERE* :

```
1 SELECT etud.no_etud, nom, adresse, dat_nais, cours.no_cours,titre, local, no_prof, note,
  note_p
2 FROM etud INNER JOIN (cours INNER JOIN suit ON cours.no_cours = suit.no_cours) ON
  etud.no_etud = suit.no_etud;
```

La logique des deux *INNER JOIN* imbriqués est plus difficile à comprendre que celle des deux jointures liées par un AND dans une clause *WHERE*.

3.2. Principe d'indépendance

Un principe à toujours garder à l'esprit lorsque l'on construit une requête SQL est le **principe d'indépendance**. Une requête respecte le principe d'indépendance lorsqu'elle est formulée de manière à **toujours donner les bons résultats** par rapport au besoin d'information et ce **indépendamment** du contenu de la base de données à un moment précis.

Par exemple, imaginons que vous vouliez obtenir la liste des étudiant.e.s ayant plus de 40 ans et que vous prépariez la requête suivante :

```
1 SELECT no_etud, nom FROM etud WHERE dat_nais < '1982-09-22';
```

Bien que cette requête retournera les bons résultats dans certains cas (par exemple, si on l'exécute le 23 septembre 2023), elle ne retournera pas tous les bons résultats si on l'exécute à plus tard. Elle ne répond ainsi pas au principe d'indépendance. Il faudrait plutôt exploiter dans cette requête la fonction **datediff()** ainsi : `datediff(now(), dat_nais)/365 > 40`.

4. Travail individuel sur les requêtes SQL

Le travail individuel sur les requêtes SQL compte pour 25% de la note du cours. Ce dernier se divise en trois sections, la première portant sur les requêtes *SELECT*, la seconde sur les requêtes *INSERT INTO*, *UPDATE* et *DELETE* et la troisième sur une réflexion quant à l'utilisation ou la non utilisation de ChatGPT pour faire le travail. La première section est découpée en trois niveaux qui correspondent à l'avancée de la matière des trois cours sur SQL (niveaux repris dans KeSQ'L fait?). Il est ainsi possible d'avancer le travail au fur et à mesure de la présentation de la matière. Le protocole est disponible sur StudiUM[1] en version PDF et Web.

[1] https://studium.umontreal.ca/course/view.php?id=250499&section=4#TPSQL

Le livrable pour ce travail est un **journal de bord** documentant les requêtes, journal de bord préparé à partir du gabarit fourni sur StudiUM[1].

5. Ressources en lien avec le cours

Matériel de cours

- *Notes de cours [cf. sci6306_cours3_notes.pdf]*

[1] https://studium.umontreal.ca/pluginfile.php/8505767/mod_resource/content/13/sci6306_tprequetes_journal_gabarit.docx

Glossaire

Équijointure

Une **équijointure** est une jointure entre deux tables basée sur l'égalité des valeurs de leur clé externe.

Paramètre (ou argument)

Un **paramètre** (ou un **argument**) dans une fonction est un élément à préciser dans la fonction. Une fonction peut nécessite**r un ou plusieurs paramètres** pour être exécutée. Par exemple, la fonction *datediff()*, qui calcule le nombre de jours entre deux dates, comporte deux arguments à préciser dans la parenthèse, soit la date de début et la date de fin. Les deux paramètres doivent être séparés par une virgule. Par exemple : *datediff(date de fin, date de début)*.

SELECT restreint

Un **SELECT restreint** est une requête SQL n'exploitant qu'une seule table de données comme source de données dans la clause FROM.