TP 2 — Lire et écrire dans une base de données

Dans ce TP, vous allez apprendre comment ajouter des informations dans une base de données SQLite.

1 Mise en place

1.1 Organisation

1) Placez-vous dans votre répertoire `Documents` et créez à l'intérieur les sous-dossiers `R105_BDD` ▸ `TP2`.

2) Ouvrez `DBeaver`, puis créez une nouvelle base de données dans le dossier `Documents` ▸ `BUT_INFO` ▸ `R105_BDD` ▸ `TP2` que vous venez de créer. Puis nommez le fichier `tp2_<nomDeFamille>.sqlite`.

3) Ouvrez l'éditeur SQL intégré dans `DBeaver` via le bouton SQL » Nouveau script SQL. Enregistrez immédiatement le fichier dans `R105_BDD` ▸ `TP2` ▸ `tp2_<nomDeFamille>.sql`.

À rendre

Pour ce TP, vous devez rendez *uniquement* le code de vos requêtes SQL. *Il ne faudra donc pas effacer les requêtes écrites au fur et à mesure pour rendre la totalité de votre travail.* Le fichier de votre code source devra être nommé `tp2_<nomDeFamille>.sql`.

Le TP contient également des questions : vous devez rédiger un compte-rendu et le rendre *obligatoirement au format PDF* avec vos réponses. Ce compte-rendu devra être nommé `tp2_<nomDeFamille>_compteRendu.pdf`.

À la fin du TP, assemblez dans une archive ZIP nommée `R105_TP2_<nomDeFamille>.zip`, les fichiers `tp2_<nomDeFamille>_<nomDeFamille>.sql` et `tp2_compteRendu.pdf`.

1.2 Création de la base de données

4) Dans la première partie de ce TP, nous allons reprendre la base de données préparée au TP 1, qui modélise de maniière simplifiée l'IUT. Afin d'obtenir une base de données propre, récupérez le code ci-dessous :

tp2_<nomDeFamille>.sql

```sql
/* Suppression de toutes les tables pour nettoyer en cas d'erreur */
DROP TABLE IF EXISTS etudiants;
DROP TABLE IF EXISTS enseignants;
DROP TABLE IF EXISTS enseignements;

CREATE TABLE IF NOT EXISTS etudiants (
  etudiant_id    INTEGER PRIMARY KEY AUTOINCREMENT, -- Clé primaire incrément automatiq.
  etudiant_nom    VARCHAR(55),
  etudiant_prenom    VARCHAR(55),
  etudiant_annne  TINYINT(1) NOT NULL,
  etudiant_telephone    INTEGER(10),
  etudiant_age    TINYINT(2) DEFAULT NULL,
  etudiant_groupe    CHAR NOT NULL
);

CREATE TABLE IF NOT EXISTS enseignants (
  enseignant_id        INTEGER PRIMARY KEY AUTOINCREMENT,
  enseignant_nom    VARCHAR(55) NOT NULL,
  enseignant_prenom    VARCHAR(55) NOT NULL,
  enseignant_heures    INTEGER(3) NOT NULL
);

CREATE TABLE IF NOT EXISTS enseignements (
  ressource_id    VARCHAR(4) PRIMARY KEY, -- Code de la ressource.
  ressource_libelle    VARCHAR(55) NOT NULL,
  enseignant_id    INTEGER(3) NOT NULL
);
```

La clause `AUTOINCREMENT` simplifie grandement la manipulation des clés primaires qui sont de type `INGEGER` : nous n'aurons pas besoin d'attribuer nous-mêmes les valeurs, SQLite s'en occupera à notre place avec un incrément automatique. On ne l'utilise que pour les clés primaires qui sont des nombres entiers, et lorsqu'aucun autre clé « naturelle » n'existe.

Clé primaire naturelle

Une clé naturelle primaire est un colonne dans une table qui apparaît spontanément pour jouer le rôle de clé primaire. C'est le cas, dans la table `enseignements`, où l'identifiant de la ressource sera simplement son code (R101, R105, R113, S101, etc.). Dans ce cas, il n'y a pas nécessité de créer une colonne spécifique pour jouer le rôle de clé primaire. La clé sera de type chaine de caractères (à cause du R ou S en première place) et il n'y a aucun sens à utiliser `AUTOINCREMENT`.

2 Au travail !

2.1 Ajout de données dans la base

Exercice 1 (La table `enseignants`). Nous allons commencer par ajouter des enseignant·es à la table `enseignants` :

tp2_<nomDeFamille>.sql

```sql
INSERT INTO enseignants(enseignant_nom, enseignant_prenom, enseignant_heures)
VALUES
  ("BOUAOUNE", "Yasmina", 470),
  ("CARTIER", "Sébastien", 450); -- La requête se termine par un point-virgule.
```

1) Copiez/collez la requête ci-dessus à la suite de votre code SQL, et exécutez-la.

2) Quel est son effet ? Dans la partie gauche de la fenêtre `DBeaver`, ouvrez la table `enseignants`, puis cliquez sur Régénérer. Examinez les données présentes dans la table. Quelles sont les valeurs prises par les attributs ?

3) Commentez les valeurs prises par l'attribut `enseignant_id`.

Syntaxe générale des insertions

Pour insérer des valeurs dans une table, la syntaxe de la requête est toujours la même :

SQL

```sql
INSERT INTO nom_de_la_table (attribut1, attribut2, …, attributn)
VALUES
  (v1_attribut1, v1_attribut2, …, v1_attributn),
  (v1_attribut1, v1_attribut2, …, v2_attributn),
  -- etc.
  (vm_attribut1, vm_attribut2, …, vm_attributn); -- Point-virgule en fin de requête
```

Il n'est pas obligatoire de préciser tous les attributs : ceux inutilisés prendront la valeur `NULL` (si c'est autorisé par la structure de la table). Les clés primaires en `AUTOINCREMENT` n'ont pas besoin d'être renseignés et sont calculés automatiquement.

4) Le nombre d'heures de l'enseignant Réda Belaiche n'est pas encore connu. Écrire une requête qui permet d'enregistrer cet enseignant en base, mais sans renseigner `enseignant_heures`.

5) Écrire le code SQL permettant d'ajouter les autres enseignant·es que vous connaissez au sein du département informatique.

6) Ajouter la requête `SELECT * FROM enseignants;` à la fin de votre code. Quel est son effet ?

7) Même question pour `SELECT enseignant_nom, enseignant_heures FROM enseignants;`

Exercice 2 (La table `etudiants`). **1)** Ajouter des entrées dans la table `etudiants`, de sorte à obtenir les valeurs suivantes :

123 etudiant id	A-Z etudiant nom	A-Z etudiant prenom	123 etudiant annee	123 etudiant telephon	123 etudiant age	A-Z etudiant groupe
1	EL AMRANI	Maïssa	1	123 456 789	[NULL]	A
2	DUPUIS	Lucas	2	987 654 321	[NULL]	C
3	GHEFSI	Maelys	2	654 789 321	[NULL]	B
4	PANTIN	Joachim	3	[NULL]	[NULL]	A
5	CABARET	Aurélie	1	[NULL]	[NULL]	C
6	MIAH	Lina	3	[NULL]	21	B
7	HACHE	Pauline	2	[NULL]	19	B

2) Ajouter la requête suivante à votre code et exécutez-la :

tp2_<nomDeFamille>.sql

```sql
INSERT INTO etudiants(etudiant_nom, etudiant_prenom, etudiant_annee, etudiant_groupe)
VALUES ("Baptiste", "LAVENTURE", 1, 'D');
```

Nous venons d'ajouter un étudiant à la base de données… en l'inscrivant dans le groupe D, qui n'existe pas ! Nous aimerions *limiter* les valeurs possibles à `A`, `B` ou `C` pour l'attribut `etudiant_groupe`. Pour empêcher ce comportement, on ajoute ce que l'on appelle une *contrainte*, avec la clause `CHECK`.

3) Reprenez la requête qui crée la table `etudiants`, et remplacer la ligne qui définit l'attribut `etudiant_groupe` par :

tp2_<nomDeFamille>.sql

```sql
etudiant_groupe    CHAR NOT NULL CHECK(etudiant_groupe IN ('A', 'B', 'C'))
```

4) Que se passe-t-il maintenant lorsque l'on souhaite ajouter un·e étudiant·e au groupe D ?

5) Ajouter de même une contrainte qui garantit que l'âge est un nombre strictement positif, et une contrainte qui assure que l'année bien de valeur 1 ou 2 ou 3.

Exercice 3 (La table `enseignements`)**.** Comme vu dans le TP 1, l'attribut `enseignant_id` est la clé étrangère vers l'identifiant de l'enseignant·e dans la table `enseignants`. On rappelle que la clé primaire est le code de la ressource (ou de la SAÉ).

Par exemple, Sébastien Cartier est en charge de la ressource R1.06 Mathématiques discrètes et son identifiant dans la table `enseignants` est 2. Donc, dans la table `enseignements`, on ajoutera la ligne :

	enseignement_id	enseignement_libelle	enseignant_id
1	R106	Mathématiques discrètes	2

.

Voici un extrait du tableau prévisionnel de services pour cette année universitaire :

Ressource	Responsable
R1.01 Initiation au développement	Pagès
R1.02 Développement d'interfaces web	Pagès
R1.06 Mathématiques discrètes	Cartier
R1.11 Bases de la communication	Loraud
R1.12 Projet professionel et personnel 1	Bouaoune
S1.01 Implémentation d'un besoin client	Deléchelle
S1.07 Portfolio 1	Bouaoune

1) Commençons par ajouter une contrainte à notre base de données, pour que SQLite prenne en compte l'existence de la clé primaire. À la fin de la requête qui crée la table `enseignants`, ajouter la ligne suivante :

tp2_<nomDeFamille>.sql

```sql
CREATE TABLE IF NOT EXISTS enseignements ( -- Requête à modifier
    /* … code déja présent … */
    FOREIGN KEY(enseignant_id) REFERENCES enseignants(enseignant_id)
);
```

2) En se basant sur le tableau prévisionnel des services, écrire une requête qui ajoute les lignes avec les bonnes informations dans la table `enseignements`.

Les contraintes d'intégrité en SQLite

SQLite ne permet pas d'ajouter *a posteriori* des contraintes d'intégrite à une table. Il faut donc avoir les contraintes correctes *dès la conception.*

D'autres moteurs de SQL plus élaborés, tels que MySQL ou PostgreSQL offrent cette possibilité. Par exemple, la requête suivante permet de créer une contrainte avec la clause `CHECK` sur la table `enseignants`, alors que la table est déjà créée :

SQL
```sql
ALTER TABLE enseignants
ADD CONSTRAINT cond_enseigants_heures -- Nom de la contrainte (au choix).
CHECK(enseignants_heures>0);
```

Si certaines lignes déjà présentes dans la base ne respectent pas cette contrainte, une erreur sera levée. Cette fonctionnalité n'est pas prise en charge par SQLite.

2.2 Sélection de données

Maintenant que notre base de données contient des informations, nous aimerions pouvoir les afficher. Avec le mot-clé `SELECT`, nous avons déjà vu comment afficher les lignes d'une table.

Nous aimerions maintenant pouvoir *filtrer* les données selon divers critères. Cela se fait avec la clause `WHERE`. Par exmeple, la requête ci-dessous renverra uniquement les étudiant·e du groupe B :

SQL
```sql
SELECT * FROM etudiants
WHERE etudiant_groupe = 'B';
```

Exercice 4 (Utilisation de la clause `WHERE`). **1)** Écrire une requête qui renvoie uniquement les étudiant·de deuxième année qui sont dans le groupe A.

2) Écrire une requête qui affiche uniquement les SAÉ (enseignements dont l'identifiant commence par la lettre `S`). Utilisez la fonction `substr` en vous aidant de la documentation pour comprendre son fonctionnement.

3) Écrire une requête qui renvoie la liste complète des enseignant·es, en triant par ordre alphabétique selon le nom de famille. On utilisera la clause `ORDER BY etudiant_nom` à placer en fin de requête.

Pour le moment, avec le mot-clé `SELECT`, nous n'avons pu afficher que des données issues d'une seule table, sans faire de croisements entre des tables différentes. Afin d'afficher toutes les informations relatives à un enseignement (son libellé, mais aussi l'identié de l'enseignant·en charge), nous aimerions obtenir une sortie de la forme :

ressource_id	ressource_libelle	enseignant_nom	enseignant_prenom
R106	Mathématiques discrètes	CARTIER	Sébastien
R101	Initiation au développement	PAGÈS	Clément
R102	Développement d'interfaces web	PAGÈS	Clément
…	…	…	…

Pour obtenir ce résultat, nous allons utiliser la clé `enseignant_id`, qui est à la fois :

la clé primaire dans la table `enseignants` ;

la clé étrangère de la table `enseignements` vers la table `enseignants`

C'est grâce à cet attribut que nous allons faire la liaison entre ces deux tables. Grâce à la commande `SELECT`, nous allons afficher les valeurs issues des deux tables, et la clause `WHERE` va nous permettre de poser une condition d'égalité entre les attributs `enseignant_id` des deux tables :

tp2_<nomDeFamille>.sql

```
1 SELECT enseignements.enseignement_id, enseignements.enseignement_libelle,
2        enseignants.enseignant_nom, enseignants.enseignant_prenom
3 FROM enseignants, enseignements
4 WHERE enseignements.enseignant_id = enseignants.enseignant_id;
```

Analysons cette requête ligne par ligne.

- Les lignes 1 et 2 indiquent les colonnes souhaitées en sortie. Comme nous allons afficher des valeurs issues de plusieurs tables, nous devons préciser, pour chaque attribut, la table dont il est issu. Cela se fait avec la syntaxe `table.attribut`. Pour des raisons de lisibilité, il est recommandé d'utiliser une ligne par table utilisée.
- La ligne 3 indique les tables interrogées.
- La dernière ligne opère les filtres. En particulier, la condition `enseignements.enseignant_id = enseignants.enseignant_id` permet de ne garder que les lignes pour lesquelles les clés primaire et étrangère correspondent.

5) Que se passe-t-il si l'on retire la clause `WHERE` de la requête ?

2.3 Liens entre ressources et compétences

À l'IUT, l'évaluation se fait par *compétences*. Six compétences sont évaluées de manière transversale : chaque compétence est évaluée à travers plusieurs enseignements, et chaque enseignements peut intervenir dans l'évaluation de plusieurs compétences. De plus, pour chaque enseignements, il est affecté un coefficient (nombre décimal) entre 0 et 1 à chaque compétence.

Voici le tableau de compétences et un extrait des coeffficients de chaque compétence par ressource :

Cptce	Libellé
C1	Réaliser un développement d'applications
C2	Optimiser des applications informatiques
C3	Administrer des syst. informatiques cplx
C4	Gérer des données de l'information
C5	Conduire un projet
C6	Travailler dans une équipe informatique

	C1	C2	C3	C4	C5	C6
R101	39%	22%				
R102	11%				17%	10%
R106		14%		17%		
R111			6%		14%	11%
R112	2%	2%	2%	2%	2%	2%
S101	40%					

Exercice 5. Créer une table `competences` ayant deux attributs : l'identifiant (clé primaire) et le libellé de la compétence. Remplissez cette table avec les six compétences indiquées.

Nous avons à ce stade toutes les informations nécessaires. Nous aimerions maintenant écrire une requête qui permette d'obtenir la liste de tous les enseignements associés aux compétences (libellé de l'enseignement, enseignant·e responsable, coefficient de chaque compétence évaluée dans cet enseignement). Pour le moment, ce n'est pas possible car il n'y a pas de clé étrangère entre les tables `enseignements` et `competences`. De plus, la relation entre les ressources et les compétence est une *relation plusieurs à plusieurs* : chaque ressource est associée à plusieurs compétences, et chaque compétence est associée à plusieurs ressource.

Dans cette situation, on n'utilise pas de clé étrangère. À la place, on crée une table annexe qui met en relation les compétences et les enseignements :

tp2_<nomDeFamille>.sql

```sql
CREATE TABLE IF NOT EXISTS relation_enseignements_cptces(
    competence_id VARCHAR(2) NOT NULL,
    enseignement_id INTEGER NOT NULL,
    enseignement_cptce_coefficient DECIMAL(3, 2) DEFAULT NULL,
    FOREIGN KEY(competence_id) REFERENCES competences(competence_id),
    FOREIGN KEY(enseignement_id) REFERENCES enseignements(enseignement_id)
);
```

Prenez le temps de bien analyser et comprendre la structure de cette table. Maintenantnt, la ressource R101 compte pour 39% de la compétence C1. Nous allons donc ajouter dans la nouvelle table `relation_enseignements_cptces` une ligne avec les valeurs `(C1, R101, 0.39)` affectées respectivement aux attributs `(competence_id, enseignement_id, enseignement_cptce_coefficient)`.

Exercice 6.

1) Écrire une requête qui ajoute la ligne précédemment étudiée à la table nouvellement créée.

2) Écrire maintenant une requête qui ajoute *toutes* les associations enseignement-compétence-coefficient à la table.

3) Exécuter la requête `SELECT * FROM relation_enseignements_cptces;`. Le résultat obtenu est-il cohérent ?

4) Pour terminer, écrire une requête qui permet d'obtenir le tableau de synthèse suivant :

enseignement_id	enseignement_libelle	enseignant_nom	competence_id	coefficient
R106	Mathématiques discrètes	CARTIER	C2	0.14
R106	Mathématiques discrètes	CARTIER	C4	0.17
R101	Initiation au développement	PAGÈS	C1	0.39
R101	Initiation au développement	PAGÈS	C2	0.22
R102	Développement d'interfaces web	PAGÈS	C1	0.11
R102	Développement d'interfaces web	PAGÈS	C5	0.17
R102	Développement d'interfaces web	PAGÈS	C6	0.1
R112	Projet professionnel et personnel 1	BOUAOUNE	C1	0.02
R112	Projet professionnel et personnel 1	BOUAOUNE	C2	0.02
R112	Projet professionnel et personnel 1	BOUAOUNE	C3	0.02
R112	Projet professionnel et personnel 1	BOUAOUNE	C4	0.02
R112	Projet professionnel et personnel 1	BOUAOUNE	C5	0.02
R112	Projet professionnel et personnel 1	BOUAOUNE	C6	0.02
R111	Bases de la communication	LORAUD	C3	0.06
…	…	…	…	…

3 Le mot de la fin

3.1 Cardinalité entre les tables

Notion de cardinalité

Considérons une base de données munie de deux tables `A` et `B`. La cardinalité de la relation entre `A` et `B` est *le nombre d'entrées dans la table `A` qui peuvent être reliées à des entrées de la table `B`*. Elle est donnée un couple de nombres entiers (a, b) où a la valeur minimale et b est la valeur maximale.

Une cardinalité est *orientée* d'une table vers l'autre. Les cardinalités peuvent existent dans les deux sens et peuvent ne pas les mêmes.

Par exemple, chaque enseignement est géré par *un·e et un·e seul·e* enseignant·e. Cardinalité entre les tables `enseignements` et `enseignants` est égale à $(1, 1)$ du côté de la table `enseignements`. À l'inverse, un·e enseignant·e peut gérer entre 0 et un nombre (théoriquement) illimité d'enseignements. La cardinalité entre les deux tables est notée $(0, n)$ du côté de la table `enseignants`. C'est la raison pour laquelle la table `enseignements` contient une clé étrangère vers la table `enseignants`.

Les cardinalités les plus compliquées sont les cardinalités *de n à n*, i.e. les situations dans lesquelles plusieurs entregistrements de la table `A` peuvent être reliés à plusieurs enregistrements de la table `B`. C'est le cas entre les tables `enseignements` et `competences`. Nous utilisons dans cette situation une table spécifique, appelée *table de relation*, qui contient les clés primaires des tables à mettre en relation. C'est ce que fait notre table `relation_enseignements_cptces`.

Exercice 7. Trouver les cardinalités des relations entre toutes les tables de notre base de données.

3.2 À retenir

À la fin de ce TP, vous devez savoir :

- créer des tables dans une base de données, en indiquant les types de chaque attribut et les éventuelles clés primaires et clés étrangères ;
- insérer des données dans une table ;
- utiliser la syntaxe `SELECT … FROM … WHERE …` pour afficher des données d'une table et les filtrer ;
- utiliser les clés primaires et clés étrangères pour afficher ensemble des données venant de tables différentes ;
- évaluer la cardinalité d'une relation entre deux tables.