DIT DAKAR INSTITUTE OF TECHNOLOGY

"Structured Query Language."

Mars, 2021

1. Bases de données

Une base de données, permet de **stocker et de retrouver** de l'information; celles-ci peuvent être de natures différentes et plus ou moins **reliées** entre elles. Ces données peuvent être stockées sous une forme très **structurées**, ou bien sous forme **peu structurées**. Une base de données peut être localisée dans un **même lieu** et sur un même support informatisé, ou **réparties sur plusieurs machines** à plusieurs endroits. Dans ce cours, nous ne nous intéresserons qu'aux bases de données relationnelles, celles dans lesquelles les données sont stockées sous forme structurée.

2. Bases de données relationnelles

Une **base de données relationnelle** est une base de données dans laquelle les données sont organisées dans des tableaux à deux dimensions appelés des relations ou **tables**. Une base de données relationnelle consiste donc en une ou plusieurs relations. Les lignes de ces relations sont appelées des nuplets **ou enregistrements**. **Les colonnes** sont appelées des attributs.

3. SGBDR: Système de Gestion de Base de Données Relationnelles

Les logiciels permettant de créer, utiliser et maintenir des bases de données relationnelles sont des **systèmes de gestion de base de données relationnelles** (SGBDR). Le SGBD est le composant permettant d'administrer, et de contrôler la base de données. Par ailleurs, le logiciel gère également tous les accès à la base de données en lecture et en écriture. Il en existe plusieurs, les plus connues étant:

Payants, non libres:

- Oracle
- SQL Server
- Microsoft Access

Gratuits, libres*:

- PostgreSQL
- SQLite (le système que nous utiliserons)
- MySQL
- MariaDB

4. Propriétés ACID

Pour décrire les fonctions et les exigences des opérations d'un SGBDR, on utilise fréquemment l'acronyme ACID, **Atomicity, Coherence, Isolation, Durability**. Les propriétés ACID garantissent qu'une transaction informatique

est exécutée de façon fiable. Même à la suite d'un crash du programme, d'un crash du système d'exploitation, ou d'une panne d'électricité, l'intégrité de la base de données demeure:

- Atomicité: transaction faite au complet ou pas du tout.
- La cohérence: chaque transaction amènera le système d'un état valide à un autre état valide.
- L'isolation: Toute transaction doit s'exécuter comme si elle était la seule sur le système.
- La durabilité: lorsqu'une transaction a été confirmée, elle demeure enregistrée.

5. Avantages et inconvénients d'un SGBD

Avantages:

- simplification de la gestion de grands ensembles de données.
- simplification et efficacité de l'accès aux données enregistrées.
- intégrité et cohérence des données
- sécurisation et protection des données
- amélioration de la disponibilité des données

Inconvénients:

- Côut (achat de licences, achat de matériel spécifique, appel à DBA)
- Inflexibilité, rigidité du schéma de données.

6. SQL: Stuctured Query Language

SQL: Structured Query Language, en français langage de requête structurée, est le langage informatique normalisé servant à l'exploitation des bases de données relationnelles. Le langage a été créé en 1974, et en 1986, SQL est devenue une norme. Le langage est reconnu par la grande majorité des systèmes de gestion de bases de données relationnelles du marché.

Les instructions SQL s'écrivent d'une manière qui ressemble à celle de phrases ordinaires en anglais. Cette ressemblance voulue vise à faciliter l'apprentissage et la lecture. Les instructions SQL sont réparties suivant 4 domaines.

6.1. DML: Data Manipulation Language (langage de manipulation de données)

```
- SELECT: selection de données depuis la bdd
- INSERT: insertion de données dans la bdd
- UPDATE: màj des données existantes dans la bdd
- DELETE: suppression de données dans une table de la bdd
- MERGE: insertion ou màj de données dans la bdd
```

6.2. DDL: Data Definition Language (langage de définition de données)

```
- CREATE: pour créer la bdd et ses objets (table, index, function, views, ...).
- DROP: suppression d'objets de la bdd
- ALTER: modification de la structure d'une bdd
- TRUNCATE: suppression de données de la table
- COMMENT: ajout de commentaires
- RENAME: renommer un objet de la bdd
```

6.3. DCL: Data Control Language (langage de contrôle de données)

```
- GRANT: accorder certaines permissions
- REVOKE: retirer certaines permissions
```

6.4. TCL: Transaction Control Language (langage de contrôle des transactions)

```
COMMIT: marquer la fin d'une transaction
ROLLBACK: annuler toutes modifications faites sur les données
SAVEPOINT: définir un point sur lequel revenir en cas de ROLLBACK
SET TRANSACTION: démarrer une transaction
```

7. SQLite

SQLite est un système de base de données qui a la particularité de fonctionner sans serveur, on dit aussi "standalone". Il est écrit en C, *un langage de programmation impératif*, et repose sur une accessibilité via le langage SQL. L'une des grandes particularités de SQLite, la base de données est intégralement stockée dans un et un seul fichier, qui est indépendant du logiciel. C'est la raison pour laquelle nous débuterons notre cours avec ce système.

7.1. SQLite, commandes dot:

```
.tables: lister les tables de la bdd
.schema: afficher la requête SQL ayant permis la creation de la table
.headers: activer l'afficher des entêtes de colonnes
.mode: définir comment afficher les tables
.read: lire les requêtes depuis un fichier
.exit: sortir de l'interpreteur sqlite
.help: afficher un message d'aide
```

7.2. SQLite, types de données:

Nous y reviendrons plus en profondeur lorsque nous verrons comment créer une table.

```
- NULL: type utilisé pour représenter l'absence de données
- INTEGER: nombre entier signé
- REAL: nombre réel
- TEXT: chaîne de caractère
- BLOB: données binaires
- BOOL: les valeurs booléennes sont stockées en utilisant les entiers 1 (vrai) et 0 (faux)
- DATE & TIME: les valeurs de ce type sont stockées en tant que TEXT ou REAL ou INTEGER
```

8. Requêtes pour la manipulation des données:

Chaque fois que vous aurez besoin d'interroger une base de donnée pour en extraire certaine information, vous devrez écrire ce qu'on appelle une **requête**. Nous parlerons ici des requêtes permettant de: **selectionner, insérer, mettre à jour et supprimer** les données.

8.1 Select

Pour pouvoir sélectionner des données issues de la base de données, vous utiliserez la commande SELECT, qui retourne des enregistrements dans un tableau de résultat. Cette commande peut sélectionner une ou plusieurs colonnes d'une table.

Voici la syntaxe complète d'une requête de sélection vous permettant de retenir dans quel ordre doivent être invoquées les différentes clauses:

```
SELECT *
FROM table
WHERE condition
GROUP BY expression
HAVING condition
{ UNION | INTERSECT | EXCEPT }
ORDER BY expression
LIMIT count
OFFSET start
```

Voyont maintenant quelques exemples un peu plus concrets:

```
sqlite> -- selectionner toutes les colonnes de la table
sqlite> SELECT * FROM nom_table;

sqlite> -- selectionner uniquement la colonne "nom_champ" de la table "nom_table"
sqlite> SELECT nom_champ FROM nom_table;

sqlite> -- selectionner plusieurs colonnes de la table "nom_table"
sqlite> SELECT nom_champ1, nom_champ2, nom_champ3 FROM nom_table;
```

Une des fonctionnalités les plus intéressantes de SQL est celle nous permettant de filtrer nos données. C'est à dire, sélectionner uniquement les données respectant certains critères. Voyons ce que sont les clauses WHERE et LIMIT.

Clause WHERE:

La clause WHERE dans une requête SQL permet d'extraire les enregistrement d'une table respectant une certaine condition. Exemples:

```
sqlite> --
sqlite> SELECT * FROM nom_table where nom_champ = 43;

sqlite> -- utilisation de l'operateur booleen AND
sqlite> SELECT * FROM nom_table WHERE nom_champ1 = 'FOOBAR' AND nom_champ2 < '1930-01-01';

sqlite> -- utilisation de l'operateur booleen OR
sqlite> SELECT * FROM nom_table WHERE nom_champ3 = 'bar' OR nom_champ3 = 'bar';

sqlite> -- utilisation de l'operateur IN
sqlite> SELECT * FROM nom_table WHERE nom_champ3 IN ('foo', 'bar', 'baz');

sqlite> -- utilisation de l'operateur LIKE
sqlite> SELECT * FROM nom_table WHERE nom_champ3 LIKE "_a%"

sqlite> -- utilisation de la clause GLOB
sqlite> SELECT * FROM nom_table WHERE nom_champ3 GLOB "?a*"
```

Clause LIMIT:

La clause LIMIT est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'ont souhaite obtenir. Examples:

```
sqlite> -- selectionner les 10 premier enregistrement
sqlite> SELECT * FROM nom_table LIMIT 10;

sqlite> -- selectionner 5 enregistrement, à partir du 11e
sqlite> SELECT * FROM nom_table LIMIT 10, 5;

sqlite> -- selectionner 10 enregistrement, à partir du 6e
sqlite> SELECT * FROM nom_table LIMIT 10 OFFSET 5;
```

NULL, COALESCE, IFNULL:

NULL est un terme utilisé pour representé **l'absence** de valeur. Un champ, ou bien, un attribut ayant la valeur NULL est un champ pour lequel aucune valeur n'a été renseignée. Ne pas confondre avec un champ de valeur "" (chaîne vide) ou un champ de valeur " " (espace).

```
sqlite> -- selectionnons toutes les colonnes du client dont l'id est égal à 59
sqlite> -- remarquez les colonnes: Company, State et Fax
sqlite> select * from customers where customerid = 59;
  CustomerId = 59
   FirstName = Puja
    LastName = Srivastava
     Company =
     Address = 3,Raj Bhavan Road
        City = Bangalore
       State =
     Country = India
  PostalCode = 560001
       Phone = +91 080 22289999
         Fax =
       Email = puja_srivastava@yahoo.in
SupportRepId = 3
sqlite>
```

```
sqlite> -- selectionnons tous les enregistrements de la table clients
sqlite> -- ayant la colonne Company non renseignée
sqlite> select * from customers where company is NULL;
```

```
sqlite> -- selectionnons tous les enregistrements de la table clients
sqlite> -- ayant la colonne Company bel et bien renseignée
sqlite> select * from customers where company is not null;
```

Il peut être assez gênant de repérer les champs non renseignés. Pour cette raison, il existe une **commande dot** nous permettant de rapidement repérer ces champs. Exécutez la commande qui suit. Puis, réexécutez les 3 requêtes ci dessus. Que remarquez vous?

```
sqlite> .nullvalue NULL
```

Une autre manière de détecter rapidement les champs dont la valeur est NULL, est l'utilisation de la fonction **coalesce**. Cette fonction prend *au moins* deux paramètres, le premier étant le nom d'une colonne, le second étant la valeur à afficher si jamais la colonne est NULL. Exemple:

```
sqlite> select
   ...>  customerid,
   ...>  firstname,
   ...>  lastname,
   ...>  coalesce(Company, "No registered company"),
   ...>  address,
   ...>  city,
   ...>  coalesce(State, "No registered State"),
   ...>  country,
   ...>  postalcode,
   ...>  phone,
   ...>  coalesce(Fax, "No registered Fax"),
   ...>  email,
   ...>  supportrepid
   ...> from customers
   ...> where customerid = 59;
```

La fonction IFNULL accepte deux arguments et retourne le premier qui est non NULL. Si les deux arguments sont NULL, alors la fonction retournera NULL.

SQLite en tant que calculatrice:

Le prompt SQLite peut en effet être utilisé comme une calculatrice. Une image valant mieux que mille paroles:

```
sqlite> select 2+2;
  2+2 = 4
sqlite>
sqlite> select 2+2 as foo;
  foo = 4
sqlite>
sqlite> select (((10-4)*(5+6))-20);
(((10-4)*(5+6))-20) = 46
sqlite>
sqlite> select 42/0;
 42/0 =
sqlite>
sqlite> .nullvalue NULL
sqlite>
sqlite> select 42/0;
 42/0 = NULL
sqlite>
sqlite> select round(3.14154265359, 2) as pi;
   pi = 3.14
sqlite>
```

Fonctions d'aggrégations:

Une fonction d'aggrégation effectue un calcul sur une ou plusieurs valeurs *(d'une même colonne)* et ensuite retourne un résultat. Les fonctions d'aggrégations sont les suivantes:

- AVG – calcule la moyenne
- COUNT – compte le nombre de lignes
- MIN – renvoie le minimum
- MAX – renvoie le maximum
- SUM – calcul la somme

Exemples:

```
sqlite> -- selection de la somme de toutes les valeurs dans la colonne total de la table invoice
sqlite> select sum(total) as somme_totale from invoices;

sqlite> -- selection du nombre total de valeurs dans la colonne total de la table invoice
sqlite> select count(total) as nombre_total_factures from invoices;

sqlite> -- selection de la moyenne de toutes les valeurs stockées dans la colonne total
sqlite> select avg(total) as average_invoice_amount from invoices;

sqlite> select min(total) from invoices;

sqlite> select max(total) from invoices;
```

8.2 INSERT

La requête INSERT est utilisée lorsque nous avons besoin d'insérer de nouveaux enregistrements dans la base de donées. Il existe différentes manières d'insérer des enregistrements dans une table. Voici la syntaxe complète d'une requête dinsertion:

```
INSERT INTO table (colonne1, colonne2 ,..)
VALUES( valeur1, valeur2, ...);
```

Voyons maintenant quelques exemples un peu plus concrets:

Insérer un seul enregistrement:

```
sqlite> -- insérer un nouvel artiste dans la table
sqlite> INSERT INTO artists (name) VALUES('Omar Pene');
```

La champ `ArtistId` est un champ auto incrémenté. Cela signifie qu'une valeur pour ce champ sera générée puis insérée automatiquement par Sqlite. C'est la raison pour laquelle, nous pouvons nous contenter d'insérer uniquement le nom de l'artiste.

Insérer plusieurs enregistrements:

```
sqlite> -- insérer plusieurs artistes dans la table
sqlite> INSERT INTO artists (name)
VALUES
    ("Cheikh Lô"),
    ("Baaba Maal"),
    ("Souleymane Faye");
```

Insérer des valeurs par défaut:

Il est possible à la création d'une table, de spécifier des valeurs par défaut pour une colonne (nous y reviendrons). Par conséquent, il est possible de dire à Sqlite d'insérer un enregistrement en utilisant les valeurs par défaut:

Si la valeur par défaut n'est pas définie, et si la colonne ne possède pas la contrainte `Not NULL` (nous y reviendrons), alors la valeur insérée par défaut sera la valeur: `NULL`:

```
sqlite> -- insérer les valeurs par défaut
sqlite> INSERT INTO artists DEFAULT VALUES;
```

Vérifiez qu'un enregistrement a bel et bien été inséré dans la table.

Insérer des valeurs venant d'une requête SELECT:

```
sqlite> -- selectionner les champs firstname et lastname du client ayant l'id 59
sqlite> -- concaténer le résultat de cette sélection
sqlite> select firstname || ' ' || lastname from customers where customerid = 59;
Puja Srivastava
sqlite> --
sqlite> insert into artists (name)
   ...> select firstname || ' ' || lastname from customers where customerid = 59;
sqlite>
```

Pour chaque enregistrement retourné par la requête SELECT, une nouvelle ligne sera inserée dans la table. Attention, le nombre de colonnes dans la requête SELECT doit être égal au nombre de colonnes dans la requête INSERT. Vous vérifiez que les insertions ont toutes été effectuées correctement.

8.3 UPDATE

Maintenant que nous savons sélectionner et insérer de nouveaux enregistrements dans une table, voyons comment effectuer une mise à jour.

La commande UPDATE est celle qui est utilisée pour changer des valeurs existantes dans une table. Il est possible de mettre à jour une ou plusieurs colonnes de la même table. Il est possible de mettre à jour un ou plusieurs enregistrements de la même table.

Voici la syntaxe (incomplète) d'une requête UPDATE:

```
UPDATE table SET colonne1 = nouvelle_valeur [, ...] [WHERE expression];
```

Tentons d'effectuer la mise à jour du client (customers) dont l'id (customerid) est 59.

```
sqlite> -- modifier le firstname du client ayant customerid égal à 59
sqlite> update customers set firstname="foobar" where customerid = 59;
```

8.4 DELETE

La requête DELETE est utilisée pour supprimer des enregistrements dans la table. N'oubliez pas d'utiliser une clause WHERE pour spécifier les enregistrement à supprimer. Sans clause WHERE, tous les enregistrements de la table seront supprimés. Example:

```
sqlite> -- suppression des enregistrements respectant la condition
sqlite> DELETE FROM table [WHERE condition];
```

```
sqlite> -- suppression de tous les enregistrements de la table
sqlite> DELETE FROM table;
```

9. Requêtes pour la définition des données:

Dans cette section nous parlerons des requêtes permettant de: **créer, modifier, supprimer** les objets (la structure) de la base de données.

9.1 CREATE

La commande CREATE TABLE nous permet de créer de nouvelles tables dans une base de données. Nous n'en parlerons pas ici, mais cette commande permet également de créer des index, des vues et des procédure stockées.

La syntaxe est la suivante:

```
CREATE TABLE [IF NOT EXISTS] nom_table (
    colonne1 type1 PRIMARY KEY,
    colonne2 type2 NOT NULL,
    colonne3 type3 DEFAULT 0,
    constraintes
);
```

Il existe également une syntaxe vous permettant de créer une table en vous appuyant sur un SELECT:

```
CREATE TABLE nom_table AS SELECT * FROM nom_table2;
```

Les types de données

C'est à la création (ou la modification du schéma) de la table, que le type de chaque colonne est précisé. Pour voir quels sont les types que vous pouvez attribuer à vos colonnes, regarder la section 7.2.

Les contraintes

- primary key:
- not null:
- default:
- unique:
- check:
- foreign key:

9.2 ALTER

La commande ALTER est utilisée pour effectuer certaines modifications sur la structure de la table. Vous remarquerez qu'il n'est pas possible (dans SQLite) de supprimer une colonne. SQLite n'implémente pas pleinement la commande ALTER TABLE.

NB:

> Supprimer une colonne vous reviendra à créer une nouvelle table ayant exactement les même colonnes que l'ancienne, exceptée la colonne que vous voulez supprimer. Ensuite il faudra insérer les données de l'ancienne table, dans la nouvelle. Finalement, il faudra supprimer l'ancienne table. Bien sûr, le tout devra être effectué à l'aide d'une transaction.

Renommer une table:

```
ALTER TABLE ancien_nom RENAME TO nouveau_nom;
```

Ajouter une nouvelle colonne à une table:

Lors de l'ajout d'une colonne, cette dernière est ajoutée en dernière position.

```
ALTER TABLE nom_table ADD COLUMN colonne1 type contrainte;
```

Renommer une colonne de la table:

```
ALTER TABLE nom_table RENAME COLUMN ancien_nom TO nouveau_nom;
```

9.3 RENAME

Voir la commande ALTER.

9.4 DROP

La suppression d'une table se fait avec la commande DROP TABLE:

```
DROP TABLE [IF EXISTS] nom_table;
```

10. Requêtes de type SELECT beaucoup plus avancées

10.1 GROUP BY

Pour utiliser la clause GROUP BY, vous devez obligatoirement utiliser une fonction d'aggrégation. Regardons les exemples suivants:

```
sqlite3 > -- Schéma de la table Tracks
sqlite3 > .schema tracks
CREATE TABLE IF NOT EXISTS "tracks"
(
    [TrackId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [Name] NVARCHAR(200)  NOT NULL,
    [AlbumId] INTEGER,
    [MediaTypeId] INTEGER  NOT NULL,
    [GenreId] INTEGER,
    [Composer] NVARCHAR(220),
    [Milliseconds] INTEGER  NOT NULL,
    [Bytes] INTEGER,
    [UnitPrice] NUMERIC(10,2)  NOT NULL,
    FOREIGN KEY ([AlbumId]) REFERENCES "albums" ([AlbumId])
        ON DELETE NO ACTION ON UPDATE NO ACTION,
    FOREIGN KEY ([GenreId]) REFERENCES "genres" ([GenreId])
        ON DELETE NO ACTION ON UPDATE NO ACTION,
    FOREIGN KEY ([MediaTypeId]) REFERENCES "media_types" ([MediaTypeId])
        ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_TrackAlbumId] ON "tracks" ([AlbumId]);
CREATE INDEX [IFK_TrackGenreId] ON "tracks" ([GenreId]);
CREATE INDEX [IFK_TrackMediaTypeId] ON "tracks" ([MediaTypeId]);
sqlite3 >
```

On remarque que chaque TrackId est en relation avec un AlbumId. Combien il y a de Tracks dans cette table? Il y en a 3503:

```
sqlite3 > -- appel de la fonction d'aggrégation count
sqlite3 > select count(trackid) from tracks;
| count(trackid) |
|----------------/
| 3503           |
changes:   0   total_changes: 0
sqlite3 >
```

Comment savoir quel est le nombre de tracks **pour chacun des albums**? Vous pouvez par exemple exécuter la requête suivante, plusieurs fois, en faisant varier la valeur de `albumid`:

```
sqlite3 > -- compter le nombre de tracks dans l'album 30
sqlite3 > select count(*) from tracks where albumid=30;

sqlite3 > -- compter le nombre de tracks dans l'album 7
sqlite3 > select count(*) from tracks where albumid=7;
```

Personne n'a envie d'effectuer cette tâche ennuyeuse. C'est pour répondre à ce genre de question, d'une manière beaucoup plus simple et beaucoup plus efficace que l'on utilise la clause GROUP BY. Syntaxe:

```
SELECT
    colonne1,
    fonction_aggregation(colonne2)
FROM
    table
GROUP BY
    colonne1,
    colonne2;
```

Comptons le nombre de pistes à l'intérieur de chaque album:

```
sqlite3 > select albumid, count(trackid) from tracks group by albumid;
```

10.2 ORDER BY

Pour ordonner la liste des enregistrements, en fonction du nombre total de pistes par albums, on ferait ceci:

```
sqlite3 > select albumid, count(trackid) as "number of tracks" from tracks group by albumid order by
```

la clause ORDER BY nous permet d'ordonner les résultat, dans l'ordre croissant ASC ou dans l'ordre décroissant DESC, en fonction d'une ou plusieurs colonnes. L'ordre par défaut, est l'ordre croissant, ASC

10.3 HAVING

La clause HAVING nous permet de spécifier un critère devant être respecté par un groupe. Cette clause ne s'utilise jamais sans la clause GROUP BY. Exemple, comment ne sélectionner que les albums ayant un nombre total de piste supérieur à 10:

```
sqlite3 > select albumid, count(trackid) as "number of tracks"
continue...> from tracks
continue...> group by albumid
continue...> having "number of tracks" > 10
continue...> order by "number of tracks" desc;
```

10.4 Sous requêtes

Comment sélectionner tous les titres d'un album quand on ne connait uniquement que le titre de l'album? On pourrait faire:

```
sqlite3 > -- retrouver l'id de l'album qui nous interesse
sqlite3 > select * from albums where title='Let There Be Rock';

sqlite3 > -- ensuite, selectionner les tracks ayant albumid=4
sqlite3 > select name from tracks where albumid=4;
```

Plutôt que de saisir deux requêtes, il est possible de n'en saisir qu'une seule et gagner un peu de temps:

```
sqlite3 > -- une requête combinant les deux précédentes
sqlite3 > select name from tracks where albumid = (
continue...> select albumid from albums where title='Let There Be Rock');
```

Autre exemple:

```
sqlite3 > select name from tracks where albumid in (
continue...> select albumid from albums where title glob "?a*")
continue...> limit 30;
```