DIU Enseigner l'Informatique au Lycée

Bases de données

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Plan

Au programme de la leçon

- Se familiariser avec le modèle relationnel
- Se familiariser avec le langage SQL

Le matin :

- Présentation du modèle relationnel
- SQL : créations de tables et contraintes
- Exercices

L'après-midi :

- SQL : requêtes
- SQL : mises à jour
- Exercices

Plan

Modèle ?

Système physique, mathématique ou logique représentant les structures essentielles d'une réalité et capable à son niveau d'en expliquer ou d'en reproduire dynamiquement le fonctionnement (Birou, 1966)

On passe son temps à faire ça en informatique !

On le fait en particulier (mais pas uniquement) dans le cadre **d'un système d'information** (SI)

Système d'information

Ensemble de **ressources** permettant de stocker, traîter et diffuser de l'information

Parmi les ressources :

- Infrastructure réseau
- Dispositifs d'acquisition (capteurs, postes de travails, ...)
- **Base de données**
- Logiciels (parmi lesquels les applications métiers)
- ...

Utilisation du SI

Les systèmes d'informations sont utilisés dans le cadre de processus métier.

Processus métier : ensemble cohérent de tâches effectuées par des personnes ou des équipements qui rendent un service pour un usager.

Exemple : la validation du passe navigo :

- l'usager applique son passe sur la borne
- le système vérifie la validité du titre de transport et autorise le passage

Le lien ?

Pour pouvoir informatiser des processsus métier, il faut pouvoir représenter **dans le système d'information** l'ensemble des acteurs et des tâches.

Dans notre exemple :

- Un passe (son numéro ?)
- L'abonnement relié au passe (titulaire, durée, zones, ...)
- La borne (dans quelle gare ?)
- Les gares ?
- ...

Il faut **modéliser** tout le processus et ses acteurs.

Rôle d'un modèle de données

- Représenter des entités d'un domaine d'étude (ou du monde réel)
- Représenter les relations entre ces entités
- Représenter des actions impliquant ces entités
- Représenter les **contraintes** que ces entités doivent satisfaire

Dans notre exemple

- Les usagers, les passes navigo, les gares, les zonessont des entités
- Le faite qu'un usager détient un certain passe est une relation
- Le fait de renouveler son abonnement est une action
- Le fait qu'un abonnement ne peut bénéficier à la fois des tarifs étudiants -26 ans et séniors est une contrainte

Modèle relationnel

- Introduit par Edgar Frank Codd en 1969 (chez IBM)
- les entités sont les éléments d'une **relation n-aire** (au sens mathématique du terme)
- les relations vont pouvoir être manipulées au moyen de formules logiques (algèbre relationnelle hors programme)
- les contraintes peuvent aussi s'exprimer comme des formules logiques (toujours hors programme)

Notre objectif : décrire précisément les règles du modèle et énoncer les principaux types de contraintes

Relations ?

Une **relation** est un ensemble de n-uplets.

Chaque composante d'un n-uplet est appelé **un attribut**

Chaque attribut appartient à un ensemble appelé son domaine

Le nom et l'ensemble des domaines des attributs est appelé le **schéma** de la relation

Exemple :

```
Usager(Nom : Chaine, Prénom : Chaine, datenaiss : Date, Id : Entier)

Usager = { ("Nguyen", "Kim", 31/10/1981, 123456),
           ("Kenobi", "Obi-Wan", 01/01/7920, 612314),
           ... }
```

Domaines des attributs ?

Le modèle relationnel ne définit pas exactement la liste des domaines.

La seule contrainte est qu'un domaine est **atomique** et en particulier **ne peut pas être une relation**.

Quelle conséquence ? Reprenons notre exemple :

On suppose que les bornes ont un numéro de série et la date de leur dernière opération de maintenance.

Comment **modéliser** le fait qu'un ensemble de bornes est dans une gare ?

Gare = { ("Gare du Nord", { (1234231, 01/10/2020), (2321423, 03/04/2019), … }),
("Le Guichet", { (837913, 17/12/2019), (1000473, 04/05/2018), … }),
…
}

Interdit par le modèle relationnel!

Domaines des attributs (2)

Le modèle relationnel ne fixe pas exactement les domaines des attributs.

Nous allons utiliser des types « usuels » des langages de programmation dans un premier temps

- **`String`** : chaînes de caractères (du texte)
- **`Int`** : des entiers (sans précision de taille)
- **`Float`** : des nombres flottants
- **`Boolean`** : des valeurs de vérité (vrai/faux)
- **`Date`** : un type permettant de représenter des dates

Modélisation relationnelle des données

1. Déterminer les **entités** que l'on souhaite manipuler
2. Représenter l'ensemble des entités comme des **relations** en donnant leur schéma
3. Définir les **contraintes** de la base de données, c'est à dire les propriétés logiques que les données doivent vérifier à tout moment.

Quatre types de contraintes

1. Les contraintes de domaine
2. Les contraintes d'entité
3. Les contraintes de référence
4. Les contraintes utilisateurs

On utilisera l'exemple (restreint) du passe Navigo

- Des usagers (avec des abonnements)
- Des abonnements (avec des types de forfaits, des zones, des dates de validité)
- Des passes (relié à un abonnement)
- Des bornes (dans des gares qui sont dans des zones)
- Des gares

L'action que notre modélisation doit permettre : la validation

- La borne vérifie que l'abonnement associé au passe est valide et dans la bonne zone

Contraintes de domaine

Ce sont toutes les propriétés sur les données qui peuvent être garantie en utilisant le bon type de données.

Par exemple, si une information est une date, utiliser le type **`Date`** plutot que le type **`String`** permet d'éviter les dates invalides du style « **`Douze fluxbors 29992`** ».

Usager (nom : String, prenom : String, datenaiss : Date,
adresse : String, cp : String, ville : String,
email : String, tel : String)

Ici peu d'autres choix possibles :

`Int` pour le code postal ou le téléphone ? attention aux 0 significatifs

Contrainte d'entité

Règle fondamentale : **UN** élément d'une relation doit être associé à exactement **UNE** entité que l'on modélise.
Dans le modèle relationnel, les relations sont des ensembles ⇒ absence de doublons.

Une **clé primaire** d'une relation est un sous-ensemble d'attributs qui identifie chaque entité de façon **unique**.

Usager (nom : String, prenom : String, datenaiss : Date,
adresse : String, cp : String, ville : String,
email : String, tel : String)

Donner des clés primaires pour la relation Usager ?

- Le nom ? NON, deux usagers peuvent avoir le même nom
- Le nom et le prenom ? NON, pour les même raisons
- L'email ? NON, **famille.michu@mail.com**, ... sauf si processus informatisé qui vérifie l'unicité (enregistrement par email)
- Le nom, le prenom, la date de naissance et l'email ? Ça semble raisonnable

Clé primaire

L'information de clé primaire **fait partie du schéma**.
La notation standard consiste à souligner les attributs faisant partie de la clé primaire :

Usager (nom : String, prenom : String, datenaiss : Date,
adresse : String, cp : String, ville : String,
email : String, tel : String)

La clé primaire force une contrainte d'unicité sur les entités :

$$\forall(n_1, p_1, d_1, a_1, c_1, v_1, e_1, t_1) \in \text{Usager}, (n_2, p_2, d_2, a_2, c_2, v_2, e_2, t_2) \in \text{Usager}$$

$$(n_1 = n_2 \wedge p_1 = p_2 \wedge d_1 = d_2 \wedge e_1 = e_2) \Longrightarrow (a_1 = a_2 \wedge c_1 = c_2 \wedge v_1 = v_2 \wedge t_1 = t_2)$$

Contrainte de référence

- On souhaite représenter une association entre deux entités (ou plusieurs)
- On doit rester dans le cadre du modèle relationnel

Exemple : Un usager a un contrat d'abonnement

On commence par modéliser les abonnements :

Abo(num : Int, type : String, datfin : Date, zonedeb : Int, zonefin : Int)

Est-ce pertinent ? Est-ce qu'on aurait pu juste rajouter ces attributs à Usager ?

- Est-ce qu'un usager peut ne pas avoir d'abonnement ? Si oui, alors que faire si tout est dans une seule table ?
- Est-ce qu'un usager peut avoir deux abonnements ? trois ? un nombre arbitraire ?
- Quand un abonnement arrive a échéance, est-ce qu'on veut le conserver ? Abonnement annuel renouvelé ⇒ mise à jour de la date. Abonnement hebdomadaires irréguliers ?

Clé étrangère

Une **clé étrangère** est un ensemble d'attributs d'une relation R qui sont une clé primaire dans une autre relation S.
Pas de notation standard, on utilise le soulignement pointillé plus une indication en français:

Abo_Usager(num : Int, nom : String, prenom : String,
datenaiss : Date, email : String)

avec num clé primaire de Abo
(nom, prenom, datenaiss, email) clé primaire de Usager

La relation Abo_Usager permet de faire le lien entre un numéro d'abonnement et un usager.

- On peut avoir un usager sans abonnement ou avec plusieurs
- On peut avoir un abonnement sans usager ou avec plusieurs
- Un usager qui a un abonnement a forcément un abonnement existant
- Un abonnement associé à un usager l'est forcément à un usager existant

Autres contraintes de cardinalité

Les notions de clés primaires et clés étrangères, on peut forcer d'autres contraintes de cardinalité:

num clé primaire de Abo
(nom, prenom, datenaiss, email) clé primaire de Usager

Abo_Usager(num : Int, nom : String, prenom : String, datenaiss : Date, email : String)

⇒num ne peut apparaître qu'une fois. Un abonnement ne peut pas être associé à deux personnes.

Abo_Usager(num : Int, nom : String, prenom : String, datenaiss : Date, email : String)

⇒Un usager ne peut apparaître qu'une fois, un a au plus un abonnement

Abo_Usager(num : Int, nom : String, prenom : String, datenaiss : Date, email : String)

⇒Un usager a au plus un abonnement, tous les usagers ont des abonnements distincts

Contraintes utilisateur

Il s'agit de contraintes qu'on appelle parfois *contraintes métiers*, liées au domaine d'application considéré

- Une date de fin est raisonnable
- Le type d'abonnement n'est pas une chaîne arbitraire (Jour, Hebdomadaire, Mensuel, Annuel)
- Le code postal doit référencer une commune existante
- ...

Ce sont souvent les plus importantes. On préconise de les saisir en français de façon précise. Dans le modèle relationnel de Codd, on pourrait les spécifier en logique du premier ordre (évidemment hors programme).

Plan

SGBD

Système de Gestion de Bases de Données :

- c'est un logiciel
- d'acquisition des données (ajout, mise à jour, suppression)
- d'interrogation (requêtes)
- avec des droits d'accès (qui peut voir quelles données)
- permettant des accès concurrents aux données
- permettant la spécification et imposant le respect des contraintes

Un SGBD utilisant le modèle relationnel est un SGBD relationnel.
Quelques noms de SGBDR :

- PostgreSQL (logiciel libre)
- MariaDB (ex. MySQL, logiciel libre)
- Oracle
- SQL Serveur (Microsoft)
- DB2 (IBM)

SQL

Le langage standard qui permet d'intéragir avec un SGBDR

- Langage de modélisation de données (relations, contraintes)
- Langage de requêtes
- Langage de contraintes
- Gestion des droits d'accès
- Langage de programmation

Standardisé : ISO/IEC 9075, dernière version 2016.

Tous les systèmes implémentent **une partie du standard**

Tous les systèmes implémentent **leurs propres extensions**

⇒ difficile d'écrire du SQL qui fonctionne sur tous les systèmes

Caractéristiques du langage SQL

- Déclaratif : on dit ce qu'on veut faire, pas comment
- Fortement typé : toutes les expressions ont un type unique
- Insensible à la casse

Vocabulaire :

- Table : relation
- Colonne : attribut
- Base de données (database) : ensemble de tables

Création de tables

```
CREATE TABLE MaTable (
  att1 type1 [constr_col1], …, attn typen [constr_coln]
  [, constr_table]);
```

- **MaTable** : nom de la table
- ***att$_i$*** : nom de l'attribut *i*
- ***type$_i$*** : type de l'attribut *i*. Exemples de types: **INTEGER**, **VARCHAR(*n*)**, ... (dépend du système utlisé)
- ***constr_col$_i$*** : contrainte sur la colonne *i*. Exemple de contraintes: **PRIMARY KEY**, **NOT NULL**, **DEFAULT *n*, …**
- ***constr_table*** : contrainte de table. Exemple de contrainte de table: **CHECK *cond*, UNIQUE (*col1*, …, *coln*)**, ...

Les types numériques

- **`SMALLINT`** : les entiers 16 bits signés
- **`INT`** ou **`INTEGER`** : les entiers 32 bits signés
- **`BIGINT`** : les entiers 64 bits signés
- **`DECIMAL(`*`t`*`, `*`f`*`)`** : décimal signé de *t* chiffres dont *f* après la virgule
- **`REAL`** : flottant simple précision (32 bits)
- **`DOUBLE PRECISION`** : flottant double précision (64 bits)

`DECIMAL` est particulièrement imporant dans un contexte monétaire (banque, transactions, ...) car exact.

Les types de chaîne de caractères

- **`CHAR(`*n*`)`** : types des chaînes de caractères d'exactement *n* caractères. Si des caractères sont manquant, ils sont complétés par des espaces.
- **`VARCHAR(`*n*`)`** : types des chaînes de caractères d'au plus *n* caractères.
- **`TEXT`** : types des chaînes de caractères de taille arbitraire.

Les chaînes de caractères sont délimitées par « `'` »

```
'Bonjour, ça va ?'
'C''est moi!'
```

On échape un `'` en en mettant deux. Pour le reste (**`\n`** par exemple) ce n'est pas standard.

Les booléens

Il existe un type `BOOLEAN` qui est inégalement supporté par les différents systèmes. S'il n'est pas supporté, on peut utiliser `CHAR(1)` et deux chaines `'T'` et `'F'`.

Les dates et les durées

- **`DATE`** : une date au format **`'AAAA-MM-JJ'`**.
- **`TIME`** : une heure au format **`'hh:mm:ss'`**.
- **`DATETIME`** : un instant au format **`'AAAA-MM-JJ hh:mm:ss'`**.

Les types sont beaucoup plus complexes que ça. Ils gèrent les années bisextiles, les fuseaux horaires,... On peut calculer précisément des décalages (par exemple *d* **`+ 10`** ajoute 10 jours à la date *d*).

Type NULL

Le type `NULL` ne contient qu'une valeur, la valeur `NULL`.
Elle représente une absence de valeur.
(On précisera le comportement plus tard, en présentant les requêtes).

Exemple de création de tables

```
CREATE TABLE Usager(nom VARCHAR(255),
                    prenom VARCHAR(255),
                    datenaiss DATE,
                    adresse VARCHAR(255),
                    cp CHAR(5),
                    ville VARCHAR(50),
                    email VARCHAR(100),
                    tel VARCHAR(20));

CREATE TABLE Abo(num INT,
                 type VARCHAR(10),
                 datefin DATE,
                 zonedeb SMALLINT,
                 zonefin SMALLINT);
```

Que manque-t-il ? les contraintes !

Contraintes en SQL

Les contraintes peuvent être spécifiées après le type, pour chaque colonne, ou globalement après la dernière colonne (en particulier lorsqu'elles mettent en jeu plusieurs colonnes).

- **`PRIMARY KEY`** : contrainte de clé primaire
- **`REFERENCES t(c)`** : contrainte de clé étrangère. La colonne concernée est une clé étrangère faisant référence à la colonne *c* de la table *t*.
- **`NOT NULL`** : la valeur **`NULL`** n'est pas autorisée dans la colonne. **`PRIMARY KEY`** implique toujours **`NOT NULL`**.
- **`UNIQUE`** : les valeurs de la colonne doivent être unique.

Contraintes multi-colonnes

- **PRIMARY KEY(col_1, …, col_k)** : les colonnes mentionnées constituent ensemble la clé primaire de la table.
- **FOREIGN KEY (col_1, …, col_k) REFERENCES *t*(col'_1, …, col'_k)**: les colonnes col_i constituent une clé étrangère. Elles correspondent aux colonnes col'_i de la table ***t***
- **UNIQUE(col_1, …, col_k)** : les colonnes mentionnées doivent former un n-uplet unique dans la table.
- **CHECK (*e*)** : l'expression booléene *e* doit être vérifiée (cf exemples)

Exemple de création de tables avec contraintes

```
CREATE TABLE Usager(nom VARCHAR(255),
                    prenom VARCHAR(255),
                    datenaiss DATE,
                    adresse VARCHAR(255),
                    cp CHAR(5),
                    ville VARCHAR(50),
                    email VARCHAR(100),
                    tel VARCHAR(20),
                    PRIMARY KEY (nom, prenom, datenaiss, tel),
                    CHECK (email LIKE '%@%'));

CREATE TABLE Abo(num INT PRIMARY KEY,
                 type VARCHAR(12),
                 datefin DATE,
                 zonedeb SMALLINT,
                 zonefin SMALLINT,
CHECK (zonedeb >= 1 AND zonefin <= 5 AND zonedeb < zonefin),
CHECK (type = 'mensuel' OR type = 'hebdomadaire' OR type = 'annuel'
       OR type = 'journalier'),
CHECK (num >= 0));
```

Revenons sur notre modélisation

```
CREATE TABLE Abo_Usager (num INT REFERENCES Abo,
                         nom VARCHAR(255),
                         prenom VARCHAR(255),
                         datenaiss DATE,
                         email VARCHAR(100),
                         FOREIGN KEY (nom, prenom, datenaiss, email)
                         REFERENCES Usager);
```

Quel souci avec cette modélisation ?

Des informations sont dupliquées dans deux tables :

- Redondance : on stocke deux fois une information, perte d'espace
- Traitement complexe : comment mettre à jour l'email ?
  - Il ne faut pas oublier de mettre à jour toutes les occurrences (anomalie de mise à jour)
  - Il faut dupliquer la ligne dans **`Usager`** avec juste l'**`email`** de différent ...
  - puis mettre à jour la ligne dans **`Abo_Usager`** ...
  - puis supprimer la ligne originale dans **`Usager`**

Comment faire ?

La pratique usuelle consiste à associer un identifiant numérique, à chaque ligne, qui devient la clé primaire

```
CREATE TABLE Usager(uid INT PRIMARY KEY,
                    nom VARCHAR(255) NOT NULL,
                    prenom VARCHAR(255) NOT NULL,
                    datenaiss DATE NOT NULL,
                    adresse VARCHAR(255),
                    cp CHAR(5),
                    ville VARCHAR(50),
                    email VARCHAR(100) NOT NULL,
                    tel VARCHAR(20),
                    UNIQUE (nom, prenom, datenaiss, email),
                    CHECK (email LIKE '%@%'));
```

La contrainte **UNIQUE** permet de conserver l'unicité du 4-uplet que l'on avait avec la clé primaire.

Suite de l'exemple

```
CREATE TABLE Abo_Usager (num INT REFERENCES Abo,
                         uid INT REFERENCES Usager);
```

La mise à jour de l'email est simple (on modifie la table **`Usager`**)

On déporte le problème sur la mise à jour de l'**`uid`** mais ce n'est pas censé se produire (c'est un identifiant interne).

À propos des identifiants

On aimerait avoir un type de compteur permettant de choisir un identifiant unique (car ça doit être une clé)

Il y a des extensions propriétaires (type `SERIAL` en PostgreSQL), et un type `SEQUENCE` dans le standard, inégalement supporté.

Il faut voir en fonction des systèmes utilisés, ou saisir les identifiants « à la main »

Insertion dans une table

```
INSERT INTO t VALUES (e1, …, en);
```

Insère dans la table ***t*** les valeurs des expressions $\mathbf{e_i}$ données dans l'ordre de déclaration des colonnes.

```
INSERT INTO Usager VALUES (1, 'Nguyen', 'Kim', '1981-10-31',
                   '14 rue du yahourt', '91400', 'Orsay',
                   'kn@lri.fr', '012345678');

INSERT INTO Usager VALUES (2, 'Kenobi', 'Obi-Wan', '7920-01-01',
                   '10 rue des étoiles', '91940', 'Les Ulis',
                   'benkenobi@caramail.fr', '01000000');
```

Insertions et contraintes

```
INSERT INTO Usager VALUES (1, ...);
ERROR:  duplicate key value violates unique constraint "usager_pkey"
DETAIL:  Key (uid)=(1) already exists.

--

INSERT INTO Usager VALUES (3, 'Nguyen', 'Kim', '1981-10-31',
                    '14 rue du yahourt', '91400', 'Orsay',
                    'abc', '012345678');
ERROR:  new row for relation "usager" violates check
constraint "usager_email_check"

--

INSERT INTO Usager VALUES (3, 'Nguyen', 'Kim', '1981-10-31',
                    '14 rue du yahourt', '91400', 'Orsay',
                    'kn@lri.fr', '012345678');
ERROR:  duplicate key value violates unique constraint
 "usager_nom_prenom_datenaiss_email_key"
```

Insertions et contraintes (suite)

```
INSERT INTO Abo VALUES(1,'mensuel', '01-01-2021', 1, 5);
INSERT INTO Abo_Usager VALUES(1,1);

INSERT INTO Abo_Usager VALUES(1,3);
ERROR:  insert or update on table "abo_usager" violates
foreign key constraint "abo_usager_uid_fkey"
DETAIL:  Key (uid)=(3) is not present in table "usager".
```

Suppression de tables

```
DROP TABLE t1, … , tn;
```

Supprime les tables dont le nom est donné. Attention, la suppression ne doit pas violer une contrainte de clé étrangère.

```
DROP TABLE Usager;
ERROR:  cannot drop table usager because other objects depend on it
```

Il faut donc ordonner les suppressions. Un **DROP** échouera si la table n'existe pas

Plan

Interroger les données

Les requêtes sont la raison d'être du langage SQL.

Les données sont ajoutées à la base, on souhaite en tirer du **sens**.

SQL : **Structured Query Language**.

Une requête SQL définit une intention, un résultat que l'on souhaite obtenir.

Une requête SQL ne donne jamais les algorithmes bas niveau à appliquer pour calculer le résultat.

Le SGBD essaye de trouver la façon optimale de calculer le résultat.

L'ordre SELECT (version simple)

```
SELECT  res_1, …,  res_n
FROM tab_ref_1, …,  tab_ref_m
WHERE condition_w
```

L'ordre **SELECT** renvoie une **nouvelle table** contenant les résultats.

- ***tab_ref_i*** : les noms des tables que l'on veut interroger. Toutes les tables concernées doivent être mentionnées.
- ***res_i*** : des expressions qui vont constituer les colonnes de la table résultante.
- **condition_w** : une expression booléene permettant de restreindre les lignes que l'on va considérer.

La clause **WHERE** est optionnelle.

Expressions

- Expressions numériques : constantes numériques, `+`, `-`, `*`, `/`, `%` (ou `MOD(a, b)`)
- Chaînes : constantes chaînes, `||` (concaténation), opérateur `LIKE` (comparaison avec un motif)
- Comparaisons : `<`, `<=`, `>`, `>=`, `=` et `<>`
- Opérations logiques : `AND`, `OR`, `NOT(…)`
- Noms de colonnes

Exemples simples

Lister les noms, prénoms et dates de naissance des usagers :

```
SELECT nom, prenom, datenaiss FROM Usager;
```

Lister la ville de l'usager dont le mail est **kn@lri.fr** :

```
SELECT ville FROM usager WHERE email = 'kn@lri.fr';

ville
-----
Orsay
```

Lister les téléphones des usagers habitant le 92 :

```
SELECT tel FROM usager WHERE cp LIKE '92___';
```

Lister les téléphones des usagers habitant le 92 né après le 01/01/2000 :

```
SELECT tel FROM usager WHERE cp LIKE '92___'
                         AND datenaiss >= '2000-01-01';
```

Jointures

La jointure est une opération fondamentale dans les BD relationnelles.

Elle permet recoller (virtuellement) les lignes de différentes tables ayant des attributs en commun.

Exemple : on souhaite renvoyer, pour chaque nom et prénom d'usager son type d'abonnement.

Dans notre modèle :

- Le nom et le prénom sont dans la table **`Usager`**
- Le type d'abonnement est dans la table **`Abo`**
- La table **`Abo_Usager`** fait le lien : elle associe des **`uid`** d'usager à **`num`** d'abonnements.

On souhaite pouvoir dire : considère les triplets de lignes (u, ua, a) où u est une ligne de la table **`Usager`**, a est une ligne de la table **`Abo`** et **`ua`** est une ligne de la table**`Abo_Usager`** et qui sont telles que le **`uid`** de u est le même que celui de ua et le **`num`**de **`a`** est le même que celui de ua.

Jointures (exemple)

Usager

uid	nom	prenom	…
1	Nguyen	Kim	…
2	Kenobi	Obi-Wan	…
3	Torvalds	Linus	…

Abo

num	type	datefin	…
14	annuel	2020-12-31	…
17	mensuel	2020-09-30	…

Abo_Usager

num	uid
17	3
14	1

On veut pouvoir considérer une table « virtuelle » :

uid	nom	prenom	…	num	uid	num	type	datefin	…
3	Torvalds	Linus	…	17	3	17	mensuel	2020-09-30	…
1	Nguyen	Kim	…	14	1	14	annuel	2020-12-31	…

Jointures (exemple)

```sql
SELECT Usager.nom, Usager.prenom, Abo.type
FROM Usager, Abo, Abo_Usager
WHERE Usager.uid = Abo_Usager.uid
      AND Abo.num = Abo_Usager.num;
```

On liste les trois tables dans le **FROM**

On utilise la clause **WHERE** pour indiquer la condition de jointure. C'est l'ancienne syntaxe. Avec la « nouvelle » syntaxe :

```sql
SELECT Usager.nom, Usager.prenom, Abo.type
FROM Usager
JOIN Abo_Usager ON Usager.uid = Abo_Usager.uid
JOIN Abo ON Abo.num = Abo_Usager.num
```

La nouvelle syntaxe est un peu plus « déclarative ». En particulier si on a des conditions complexe dans le **WHERE**, on sépare les conditions de jointure des autres.

Jointures (exemple)

Si on veut la même requête mais seulement pour les gens nés avant 1990 :

```
-- ancienne syntaxe
   SELECT Usager.nom, Usager.prenom, Abo.type
   FROM Usager, Abo, Abo_Usager
   WHERE Usager.uid = Abo_Usager.uid
         AND Abo.num = Abo_Usager.num
         AND Usager.datenaiss < '1990-01-01';

-- nouvelle syntaxe
   SELECT Usager.nom, Usager.prenom, Abo.type
   FROM Usager
   JOIN Abo_Usager ON Usager.uid = Abo_Usager.uid
   JOIN Abo ON Abo.num = Abo_Usager.num
   WHERE Usager.datenaiss < '1990-01-01';
```

Jointures (fin)

Les jointures sont une opération essentielle, qu'il faut maîtriser.

La modélisation va directement influencer les jointures :

- Si on retire les redondances en utilisant des identifiants, on doit utiliser des jointures pour relier les données entre elles en passant par leurs identifiants
- Si on laisse des redondances, on peut économiser des jointures (mais c'est mal)

SELECT *, alias

On peut renvoyer toutes les colonnes de la « table virtuelle » avec l'instruction **`SELECT * FROM …`** :

```
SELECT * FROM Usager;
```

Dans la clause **`FROM`**, on peut donner des alias aux noms de tables (s'ils sont trop longs). Si on veut renvoyer toutes les colonnes de la table **`Usager`** et uniquement le **`type`** de la table **`Abo`** :

```
SELECT u.* , a.type
FROM Usager u
JOIN Abo_Usager au ON u.uid = au.uid
JOIN Abo a ON a.num = au.num
WHERE u.datenaiss < '1990-01-01';
```

Aggrégats

Parfois, plutôt que de renvoyer un ensemble de ligne, on veut les aggréger : calculer la moyenne, le nombre de lignes, le max ou le min selon une colonne

```
SELECT F(c) FROM t1, …, tn WHERE e;
```

où **F** peut valoir :

- **COUNT** : **c** peut valoir ***** ou n'importe quelle colonne, renvoie le nombre de résultats
- **SUM** ou **AVG** : **c** doit être une colonne de type numérique, calcule la somme ou la moyenne.
- **MAX** ou **MIN** : **c** doit être une colonne (pas *), renvoie la plus grande ou plus petit valeur.

```
SELECT COUNT(*) FROM Usager;
```

```
SELECT MAX(datenaiss) FROM Usager WHERE cp LIKE '91___';
```

à propos de LIKE

`s LIKE m` compare la chaîne `s` au motif `m`.
Un motif est :

- Une chaîne de caractères
- où `_` indique un caractère quelconque
- où `%` indique une suite de caractères quelconque, possiblement vide

Matérialisation de résultats

On peut stocker le résultat d'une requête dans une table avec la directive **INTO** :

```
SELECT * INTO gens_du_75 FROM Usager WHERE cp LIKE '75___';

SELECT nom, prenom FROM gens_du_75;
```

Le SGBD lève une erreur si la table existe déjà. On peut la supprimer avant avec **DROP TABLE …**

Attention, la table crée n'a pas de contraintes

Requêtes imbriquées (1)

SQL identifie les requêtes renvoyant une table de une case et une colonne avec la valeur contenue à l'intérieur.
On peut ainsi imbriquer des requêtes :

```
SELECT *  FROM Usager u
WHERE u.nom = (SELECT u2.nom FROM Usager u2
                                WHERE u2.uid = 512);
```

Renvoie : toutes les colonnes des lignes de la table **`Usager`** pour lesquelle le **`nom`** est le même que le nom de l'usager d'**`uid`** 512.

Requêtes imbriquées (2)

Attention, si une sous-requête renvoie plusieurs résultats, le SGBD lèvera une erreur. Le nombre de résultat **dépend des données**

```
SELECT *  FROM Usager u
WHERE u.nom = (SELECT u2.nom FROM Usager u2
                                WHERE u2.datenaiss = '1990-02-01');
```

Si plusieurs usagers sont né le 1er février 1990, la sous-requête renvoie deux résultat, et l'expression **=** échoue.
Il y a aussi une erreur si aucun usager n'est né à cette date.
La requête s'exécute sans erreur uniquement s'il n'y a qu'un usager ayant cette date de naissance.

Opérateur IN

L'opérateur **IN** est un opérateur binaire permettant de tester si un valeur est dans le **résultat** d'une requête (qui ne doit renvoyer qu'une colonne) :

```
SELECT u.nom FROM Usager u
WHERE u.datenaiss IN (SELECT u2.datenaiss FROM Usager u
                                          WHERE cp LIKE '91___');
```

Ici la sous requête renvoie l'ensemble des dates de naissances d'usagers habitant le 91.
La requête externe affiche le nom de tous les usagers dont **la date de naissance** est dans l'ensemble renvoyé par la requête intermédiaire.

Ordre et doublons

On peut supprimer les doublons d'un résultat en utilisant la clause **DISTINCT**

.

```
SELECT DISTINCT ville FROM Usager;
```

Renvoie les villes de la table **Usager** sans doublons.
On peut aussi ordonner les résultats d'une requête selon une ou plusieurs colonnes avec la clause **ORDER BY**:

```
SELECT nom, datenaiss FROM Usager
WHERE cp = '93___'
ORDER BY nom ASC, datenaiss DESC;
```

Renvoie tous les noms et dates de naissance des usagers du 93, triés par ordre croissant de noms et, en cas d'égalité de nom, par ordre décroissant de date de naissance

Conclusions sur les requêtes

- Langage « inhabituel » (déclaratif)
- Très riche (et on a vu environs 10% de ce qui est faisable)
- Repose fondamentalement sur la notion de jointure
- Interroger les données demande de connaître la modélisation de celles-ci
- Probablement la partie la plus standard sur toutes les implémentations

Plan

Mises à jour

Les mises à jour sont des opérations similaires aux requêtes (clause **`WHERE`**) mais conceptuellement plus simples.

Elles consistent en des modifications de lignes ou des suppressions.

On verra aussi une forme plus complexe de l'ordre **`INSERT`**.

La seule difficulté est qu'une mise à jour peut violer des contraintes et donc échouer.

Suppressions de lignes

On peut supprimer des lignes d'une table grâce à l'ordre **DELETE** :

```
DELETE FROM t WHERE cond;
```

supprime toutes les lignes de la table **t** pour lesquelles la condition **cond** est vérifiées.
On peut omettre la clause **WHERE** pour supprimer toutes les lignes.
Attention : supprimer toutes les lignes est différent de supprimer la table elle même.

```
DELETE FROM Usager WHERE ville = 'Paris';
```

La ligne ci-dessus est syntaxiquement correcte, mais risque d'échouer si les **uid** de ces usagers sont toujours dans la table **Abo_Usager**

Mise à jour de lignes

On peut modifier les valeurs des colonnes pour un ensemble de lignes avec l'ordre **UPDATE**

```
UPDATE t SET c1 = e1, … ,cn = en
WHERE cond;
```

remplace les valeurs des colonnes $\mathbf{c_i}$ par les valeurs des expressions $\mathbf{e_i}$.

```
UPDATE Abo SET datefin = datefin + 30 WHERE type = 'annuel';
```

Ajout 30 jours à la date de fin de tous les abonnements annuels.

Insertion sur requête

Il est parfois utile d'ajouter à une table existante toutes les lignes renvoyées par une requête.

```
CREATE TABLE abo_perime (num INTEGER PRIMARY KEY, datefin DATE);

INSERT INTO Abo_perime (SELECT num, datefin FROM Abo
                        WHERE datefin < '2020-01-01');
```

La requête ci-dessus est différente d'un **`SELECT … INTO`** car la table **`abo_perime`** possède des contraintes.

Respect des contraintes de clé étrangères

Les clés étrangères sont un outil précieux pour créer des liens entre les tables (en particulier que l'on voudra joindre)

Elles imposent un **ordre** de mise à jour des données :

Si on veut modifier une clé étrangère, la nouvelle clé primaire correspondante doit déjà exister

Si on veut supprimer une clé primaire, la clé étrangère doit être supprimée avant.

On a donc besoin de plusieurs ordres. Exemple avec la suppression de l'abonnement de **num** 17.

```
DELETE FROM Abo_usager WHERE num = 17;
DELETE FROM Abo WHERE num = 17;
```

Conclusion

Les bases de données, et en particulier les bases de données relationnelles forment un pan important de la recherche en informatique.

On ne peut évidemment pas prétendre tout présenter en quelques leçons

La partie modélisation est importante et l'exercice est utile pour d'autres branches de l'informatique (génie logiciel, programmation).

La partie SQL est complexe à cause de la syntaxe. On peu s'appuyer sur les listes en compréhension de Python :

```
SELECT e1, …, en FROM  T1 t1, …, Tk tk WHERE c;
```

est similaire à

```
[  (e1, …, en) for  t1 in T1 …,for  tk in Tk  if c ]
```