Bases de données

1 Introduction

L'objet des bases de données (abrégé BD) est la gestion d'informations, qui peuvent se révéler nombreuses et variées dans des applications réelles.

Il s'agit de trois choses :

1. Stocker les informations de manière persistente,
2. Permettre l'ajout, la suppression et la modification de ces informations,
3. Rendre facilement (de manière aisée pour des non-spécialistes) et efficacement accessible ces informations.

On entend par facilement "de manière aisée pour des non-spécialistes". On utilise pour cela une langage d'interface simple.

Les systèmes de gestion de bases de données (abrégé SGBD) assurent le stockage physique des informations, et les opérations de création / ajout / modification et recherche, pourvu qu'elles soient formulées dans un langage adéquat, appelé **langage de requête**. Pour cela, les informations doivent être organisées dans une base de données, que l'on peut définir comme un ensemble d'informations structurées.

La structure d'une base de données est définie par ses **tables**, et les **colonnes** de ces tables. Chaque table est identifiée par son nom, et contient un nombre fixé de colonnes. Chaque colonne est identifiée par son nom, et fixe un **domaine** de valeurs possibles : entiers, chaînes de caractères, ou flottants. Ce sont les lignes de ces tables, appelées **enregistrements**, qui contiennent ces informations.

Remarque : La définition de base de données n'interdit pas la multiplicité des lignes. En pratique, on s'interdira de stocker deux fois la même ligne dans le même tableau.

Si l'on fixe l'ordre des n colonnes d'une table, et que l'on note $(D_i)_{i\in[\![1,n]\!]}$ les domaines de ces colonnes, chaque enregistrement est un n-uplet de $D_1 \times ... \times D_n$.

Le nombre de lignes n'est pas limité, et leur ordre n'est pas significatif, de sorte qu'une table puisse être vue comme un sous-ensemble de $D_1 \times ... \times D_n$, que l'on appelle aussi une relation sur $D_1 \times ... \times D_n$.

Remarque : Techniquement, il ne s'agit pas réellement d'un sous-ensemble à cause de la multiplicité possible, mais plutôt d'un multi-ensemble. Mais on évitera toujours de telles redondances via l'identification d'une clé (voir plus loin).

La gestion d'une base de données consiste donc en ces trois points :

1. la conception de la base de données, c'est à dire décider de la structure la plus adéquate pour une application donnée ;
2. l'acquisition des données et leur enregistrement dans la base ;
3. la valorisation des données, c'est à dire l'extraction d'informations pertinentes.

2 Le modèle entité/association (E/A)

Le modèle entité-association est un outil qui permet de représenter la structure d'une base de données. Il est utile en phase de conception, ou pour visualiser une base de données existante.

Exemple : On cherche à enregistrer un groupe de personnes, l'endroit où elles habitent, la capitale du pays d'où elles viennent et leur âge.

Nom	Prénom	Pays de résidence	Capitale	Date de naissance
DURAND	Paul	Pays-Bas	Amsterdam	26/09/1981
WILES	Andrew	Royaume-Uni	Londres	11/04/1953
...	...	...	...	...

Remarque : On a ici une redondance entre le pays de résidence et sa capitale : cela pose des problèmes d'espace occupé, mais surtout si on doit faire une correction (imaginons qu'un pays change de capitale !), il faut s'assurer que le changement soit fait partout. Il serait ici plus adapté de stocker dans une autre table les différents pays et leur capitale, et ne stocker alors que le pays dans la table précédente.

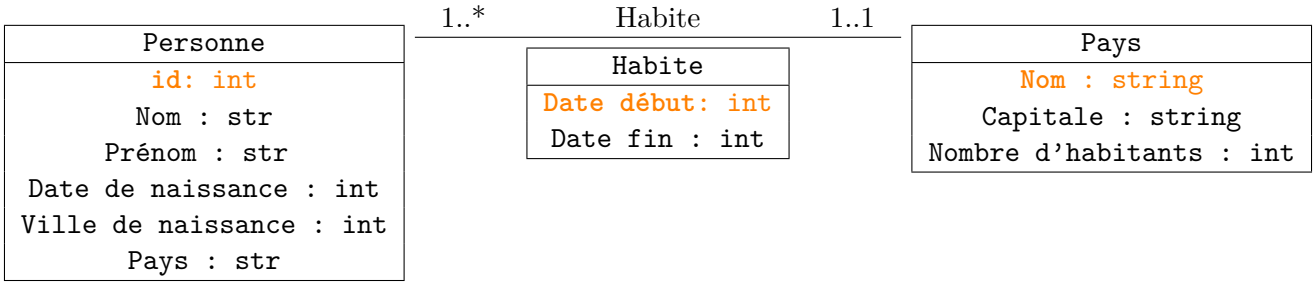
Remarque : Il faut aussi éviter les doublons, qui pourront poser des problèmes de dénombrement, de cohérence et de mise à jour. Pour cela, il faut identifier ce qui caractérise un enregistrement, c'est à dire la **clé** de cet enregistrement, et imposer que deux enregistrement aient deux clés différentes.

La clé d'une table est donc un sous-ensemble de ses colonnes, telle que deux lignes de la table ne coïncident pas sur ces colonnes. Il y a parfois plusieurs clés possibles. Celle qu'on indique au SGBD est appelée clé primaire.

Une bonne clé est :

- composée de données stables dans le temps, peu sujettes à des modifications ;
- composée de données dont on dispose pour chaque enregistrement ;
- la plus petite possible.

Si parmi les données aucune ne satisfait ces contraintes, on ajoutera un identifiant artificiel qui servira de clé.



Représentation de deux entités et d'une association entre elles. Les lignes en orange désignent les clés.

Remarque : Ici les éléments stockés en **ligne** dans la table sont réprésentés en **colonne**.

Dans le modèle entité-association, on différencie les tables qui modélisent les objets de celles qui modélisent les liens entre ces objets. Les premiers sont appelés entités, et les seconds associations.

On définit un type d'entité par la donnée de son nom, ses attributs avec leurs domaines, et sa clé, et une association binaire par son nom, les deux entités qu'elle relie (A et B), des cardinalités c_A et c_B, éventuellement des attributs. Chaque clé de A apparaît autant de fois qu'indiqué par c_A dans la table de la relation.

On notera la cardinalité c_A sur la ligne liant les deux entités, à côté de A, et c_B à côté de B (cf figure ci-dessus), de la manière suivante :

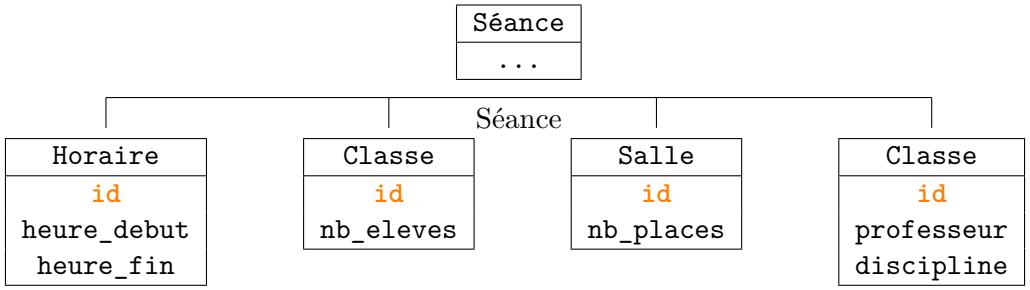
- $c_A = a..b$ signifie qu'il peut y avoir de a à b éléments de l'entité A en relation avec l'entité B. Si $a = b$, on s'autorisera à écrire $c_A = a$.
- $c_A = a..*$ signifie qu'il y a au moins a éléments de l'entité A en relation avec l'entité B. Si $a = 0$, on s'autorisera à noter $c_A = *$.

3 Simplifications

Dans le cadre du programme, on adoptera les principes suivants :

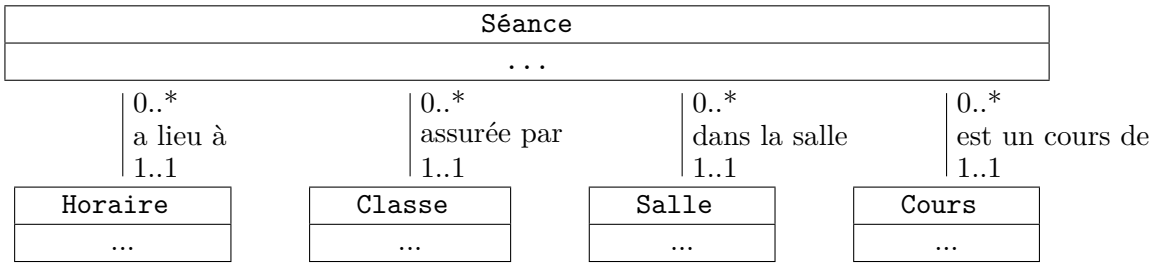
- Éviter les relations ternaires ou plus.
- Éviter les associations 0..* vers 0..*

Exemple : On cherche à gérer les différents emplois du temps des élèves et des professeurs dans un lycée. Il faut donc organiser les séances, avec un horaire précis, une classe qui aura une certaine matière avec tel professeur :



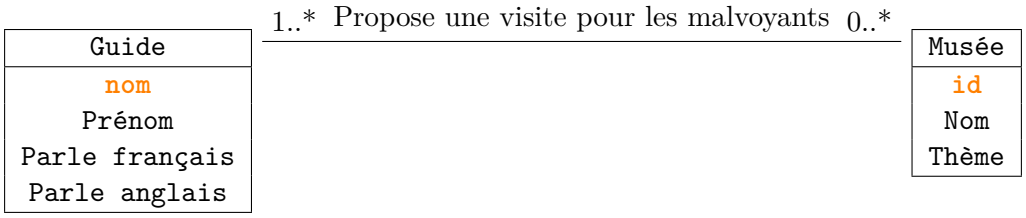
Représentation d'une séance comme association. Les cardinalités n'ont pas été représentées.

On peut alors se ramener à utiliser uniquement des relation binaires plutôt qu'une relation quaternaire :

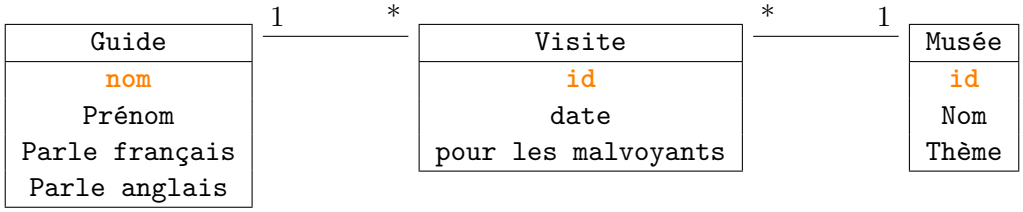


Représentation d'une séance comme entité.

Exemple : Se pose aussi la question des relations `0..*-0..*`. Prenons l'exemple d'un musée qui embauche un guide, parlant plusieurs langues. On souhaite savoir s'il est possible d'organiser des visites pour les malvoyants :



Pour éviter d'avoir à écrire cette relation, on va créer une nouvelle entité, `visite` :



4 Requêtes SQL

Une fois notre base de données créée, on va pouvoir travailler avec via le SQL.

4.1 Sélection simple

4.1.1 Agir sur les colonnes

- Le code suivant permet de récupérer certaines colonnes d'une table dans notre base de données.

```
SELECT <colonne1>,<colonne2>,<colonne3>,...,<colonneN>
FROM <table1>
```

où `<table1>` est une table de la base et `<colonne1>,<colonne2>,...<colonneN>` sont des colonnes de `<table1>`.

Remarque : Les mots-clés sont ici écrits en majuscules. Même si ça n'est pas obligatoire, c'est recommandé pour des raisons de lisibilité (les noms de colonnes/tables seront alors en minuscule).

Remarque : Si il y a plusieurs tables dans notre base de données contenant des colonnes portant le même nom, on pourra les distinguer en écrivant `tableau1.colonneA` et `tableau2.colonneB`.

- On peut récupérer les colonnes sous un nom modifié grâce au mot-clé `as` :

```
SELECT <nom_colonne> as <nouveau_nom>
FROM <table1>
```

récupérera la colonne `<nom_colonne>` sous le nom `<nouveau_nom>`.

Il est aussi possible de faire certaines opérations sur les éléments d'une même ligne (en spécifiant les colonnes de chaque élemnt), avant de récupérer le résultat :

```
SELECT <nom_colonne> + 20 AS colonnePlus20, fin-debut AS duree
```

Remarque : Si jamais les colonnes n'ont pas les mêmes tailles, les éléments manquants seront interprétés comme nuls (valeur `NULL`). Tout opérateur appliqué à `NULL` renvoie `NULL`.

Exemple : Si le jour de naissance de différents éléments est stocké dans la colonne `jour_de_naissance` de la table `personnes`, (comme un entier, mettons le nombre de jours depuis une date donnée) et que l'on a stocké le jour de leur dernier anniversaire dans la colonne `dernier_anniversaire`, on peut connaître l'âge de la personne

```
SELECT (dernier_anniversaire-jour_de_naissance)/365 AS age
FROM personnes
```

Remarque : On notera `AS` pour les colonnes, on aura tendance à l'omettre pour les tables, même si la présence ou l'absence du mot clé n'a pas d'importance.

Remarque : Pour selectionner toutes les colonnes, on peut utiliser une astérisque (*).

Le mot clé `WHERE` permet de sélectionner dans les colonnes les lignes respectant certaines conditions :

```
SELECT <colonne1>,...,<colonneN>
FROM <table1>
WHERE <cond>
```

où `<cond>` est une expression booléenne sur les attributs construite grâce aux opérateurs suivants :

- `=`
- `<>`
- `<=`
- `>=`
- `<`
- `>`
- `BETWEEN .. AND`
- `AND`
- `OR`
- `NOT`
- `IS NULL`
- `IS NOT NULL`

On peut sélectionner des lignes ayant deux valeurs distinctes sur des colonnes différentes via le mot-clé `DISTINCT` (attention cela-dit : cela supprime certains éléments, on ne sait pas lesquels !). La distinction se fera alors sur *toutes* les colonnes spécifiées, pas seulement la première.

Exemple : Si on a une table `personnes` contenant une colonne `prenom` et une colonne `pays`, et que l'on veut lister - sans doublon - les noms des personnes en France :

```
SELECT DISTINCT prenom
FROM personnes
WHERE pays == "France"
```

Exemple : Si maintenant on veut TOUS les prénoms, sans distinction de pays. Ici, deux personnes ayant le même prénom, même si elles ne sont pas nées dans le même pays, seront regroupées en un seul enregistrement contenant leur prénom.

```
SELECT DISTINCT prenom
FROM personnes
```

Exemple : Ici, si deux personnes ont le même prénom mais un pays différent, elles ne seront pas regroupées dans le même enregistrement :

```
SELECT DISTINCT prenom,pays
FROM personnes
```

4.1.2 Tronquer et trier les résultats

Le mot clé `ORDER BY` trie les lignes selon les valeurs dans la colonne juste après. En cas d'égalité, il utilise la colonne éventuellement spécifiée après pour trancher, et ainsi de suite... Il faut à chaque fois préciser si le tri se fait par ordre croissant ou décroissant, via le mot clé `ASC` ou `DESC`

Pour limiter le nombre de lignes récupérées, on peut utiliser le mot-clé `OFFSET n` qui efface les `n`-premières lignes, ainsi que le mot-clé `LIMIT k` qui ne garde que les `k` premières lignes.

```
SELECT <colonne1>,<colonne2>,...,<colonneN>
FROM <table1>
WHERE <condition>
ORDER BY <colonneA> <ASC | DESC>, <colonneB> <ASC | DESC>
LIMIT <nombre_limite>
OFFSET <decalage>
```

4.2 Agrégation

• On peut être ammené à faire des calculs opérant sur les valeurs de différentes lignes d'une même colonne, par exemple le nombre de lignes, leur moyenne, ou leur somme. Si par exemple on veut compter le nombre de personnes s'appelant Paul dans la table `personnes` :

```sql
SELECT COUNT(id)
FROM personnes
WHERE prenom == "Paul"
```

Les principales fonctions d'agrégation sont `SUM`, `AVG`, `COUNT`, `MAX`, `MIN`.

Remarque : Si l'on applique une fonction d'agrégation à une colonne, et que l'on sélectionne une autre colonne en même temps, la fonction d'agrégation s'appliquera séparément sur toutes les lignes coïncidant sur l'autre colonne.

Exemple : Si on a une table `achat` contenant les ventes d'une boutique :

identifiant	client	tarif	date
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

La requête SQL

```sql
SELECT client, SUM(tarif)
FROM achat
```

renverrra :

client	tarif
pierre	262
simon	47
marie	38
marie	38
pierre	262

• On voit sur l'exemple précédent qu'il y des doublons. Pour les éviter, il faut "grouper" les éléments par nom, via la commande `GROUP BY` (qui se place TOUJOURS après la commande `WHERE`) :

```sql
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
```

Renverra :

client	tarif
pierre	262
simon	47
marie	38

Si l'on veut alors appliquer encore appliquer un filtre, on ne peut pas utiliser `WHERE`. La commande à utiliser est `HAVING`, qui fonctionne comme `WHERE`.

4.3 Jointure

• Reprenons l'exemple des personnes et de leur pays de résidence. Si l'on veut connaître la capitale du pays dans lequel vivent les différentes personnes, on procèdera comme suit :

```sql
SELECT pers.nom, pers.prenom, pays.capitale
FROM personne AS pers
JOIN pays
ON pers.pays = pays.capitale
```

4.4 Jointure gauche

• Imaginons que l'on gère une liste de clients enregistrés dans une table `clients`, avec la date de leur dernier achat dans une autre table `achats`. Certains clients n'auront pas encore fait d'achats. Si l'on veut cependant obtenir tous les clients en y joignant leurs achats, on ne récupérera pas ces clients ! La solution est d'utiliser `LEFT JOIN` :

```sql
SELECT *
FROM clients
LEFT JOIN achats ON clients.id == achats.id_client
```

Il est alors possible de récupérer tous les clients n'ayant pas effecué d'achats :

```sql
SELECT clients.id, prenom, nom, achats.id_client
FROM clients
LEFT JOIN achats ON clients.id == achats.id_client
WHERE achats.id_clients IS NULL
```

Remarque : On utilise bien ici `IS NULL` et non `== NULL`, car `NULL` désigne l'absence de valeur : il n'y a donc rien à comparer !