# Les bases de données

## Qu'est-ce qu'une base de données ?

Une base de données (que nous nommerons BDD par commodité) est une **collection d'informations organisées afin d'être facilement consultables, gérables et mises à jour**. Au sein d'une database, les données sont organisées en lignes, colonnes et tableaux.

Une base de données est une entité dans laquelle il est possible de stocker des données de façon structurée et avec le moins de redondance possible.
Ces données doivent pouvoir être utilisées par des programmes, par des utilisateurs différents.
Ainsi, la notion de base de données est généralement couplée à celle de réseau, afin de pouvoir mettre en commun ces informations, d'où le nom de base.
On parle généralement de système d'information pour désigner toute la structure regroupant les moyens mis en place pour pouvoir partager des données.

## *Structure d'une base de données.*


## *Les SGBD*

Une base de données permet d'enregistrer des données de façon organisée et hiérarchisée.

Un **Système de Gestion de Base de Données** (SGBD) est un logiciel qui permet de stocker des informations dans une base de données. Un tel système permet de lire, écrire, modifier, trier, transformer ou même imprimer les données qui sont contenus dans la base de données.
Quelque SQBD connu :

- **MySQL** : libre et gratuit, c'est probablement le SGBD le plus connu. Nous l'utiliserons dans cette partie ;
- **PostgreSQL** : libre et gratuit comme MySQL, avec plus de fonctionnalités mais un peu moins connu ;
- **SQLite:** libre et gratuit, très léger;
- **Oracle** : utilisé par les très grosses entreprises ; sans aucun doute un des SGBD les plus complets, mais il n'est pas libre et on le paie le plus souvent très cher ;
- **Microsoft SQL Server** : le SGBD de Microsoft.
- **Google AppEngine**

Il existe deux types de SGBD.

- SGBD relationnel : les données sont représentées dans différents tableaux pouvant être liés entre eux.

- SGBD NoSQL (clé-valeur, orienté graphe, orienté document...) : les données ne sont pas structurées en tableaux mais sur des structures différentes :

  - clé-valeur : par exemple un dictionnaire qui à chaque mot (clé) associe une définition (valeur)
  - orienté graphe : associe à chaque élément les éléments liés (ex : les amis d'une personne)

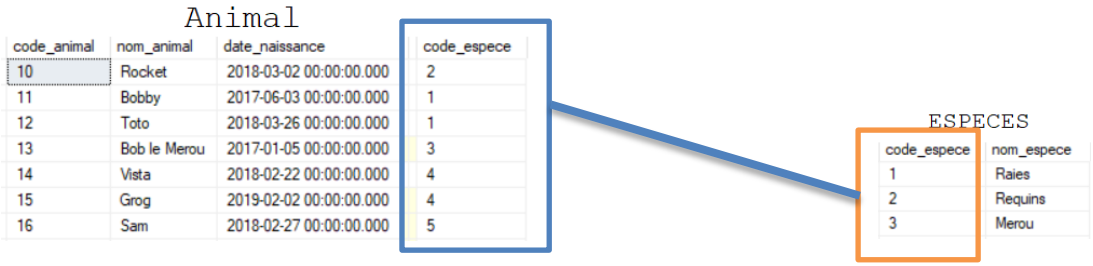
Nous allons nous intéressez essentiellement au Systéme de gestion de base de données relationnels.

Les SGBD relationnels (**SGBD-R**) sont, de loin, le type de SGBD le plus couramment utilisé lorsque l'on parle de bases de données. Ils sont basés sur un modèle relationnel, tel que nous l'avons vu dans la partie précédente de ce cours.
Pour interagir avec un SGBD-R on utilise un langage appelé **SQL***(Structured Query Language)*. Ce langage permet **d'ajouter, modifier ou supprimer des données mais aussi d'interroger la base de données** selon certains critères et faire des recoupements d'information en suivant les relations entre les tables.

# Les tables

Dans une base de données relationnels **les données sont organisées en table**, ces tables peuvent être relié entre elles, d'où le nom relationnel.

C'est-à-dire que les données peuvent avoir des **relations** entre elle. Par exemple ci-dessous, la table Animal à une relation avec la table Espèce, c'est le code espèce qui est le dénominateur commun entre les deux tables.



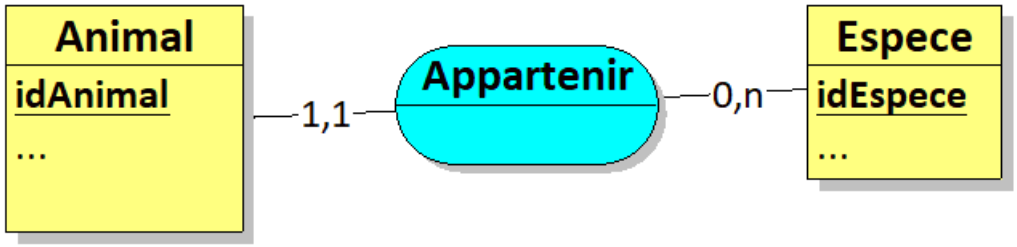
Animal

| code_animal | nom_animal | date_naissance | code_espece |
|---|---|---|---|
| 10 | Rocket | 2018-03-02 00:00:00.000 | 2 |
| 11 | Bobby | 2017-06-03 00:00:00.000 | 1 |
| 12 | Toto | 2018-03-26 00:00:00.000 | 1 |
| 13 | Bob le Merou | 2017-01-05 00:00:00.000 | 3 |
| 14 | Vista | 2018-02-22 00:00:00.000 | 4 |
| 15 | Grog | 2019-02-02 00:00:00.000 | 4 |
| 16 | Sam | 2018-02-27 00:00:00.000 | 5 |

ESPECES

| code_espece | nom_espece |
|---|---|
| 1 | Raies |
| 2 | Requins |
| 3 | Merou |

Ces relations sont souvent schématisées de cette manière :




Un animal appartient à une seule espèce alors qu'une espèce peut appartenir à plusieurs animaux (Il peut avoir plusieurs Animaux de type Félins).

Exemple : On créé une table « Etudiants » , voici sa structure :

| Numero_etudiant | nom | prenom | age |
|---|---|---|---|
| 1 | Zidane | Zinedine | 44 |
| 2 | Perrin | Loic | 33 |
| 3 | Umtiti | Samuel | 24 |

Nom des champs (ou colonnes) de la table

Données contenues dans la table

Les données peuvent être de plusieurs types, quelques exemples :

- varchar(nombre de caractères souhaité) ➔ chaine de caractères

- int ➔ nombre entier
- float ➔ chiffre à virgule
- texte ➔ chaine de caractères
- bit ➔ 0 ou 1
- datetime ➔ date avec l'heure (ex : 01/02/2018 12 :00 :05)
- date ➔ juste la date (15/04/2018)

Par exemple le Numéro_etudiant serait de type INT, les colonnes « nom » et « prenom » de type varchar(150) et « age » de type INT.

### *Les clés primaire dans les tables*

*La clé primaire d'une table est une contrainte d'unicité, composée d'une ou plusieurs colonnes, et qui permet d'identifier de manière unique chaque ligne de la table.*

Examinons plus en détail cette définition.

- **Contrainte d'unicité :** ceci ressemble fort à un index UNIQUE.
- **Composée d'une ou plusieurs colonnes :** comme les index, les clés peuvent donc être composites.
- **Permet d'identifier chaque ligne de manière unique :** dans ce cas, une clé primaire ne peut pas être NULL.

Reprenons la table suivante :

| Identifiant_unique | Nom | Prenom |
|---|---|---|
| 1 | Dupont | Bob |
| 2 | Dupont | Bob |
| 3 | Ruffier | Stéphane |

Imaginez que quelqu'un ait le même nom de famille que vous, le même prénom. En dehors de la photo et de la signature, quelle sera la différence entre vos deux cartes d'identité ? Son numéro ! C'est pareil pour une table SQL !

# La commande SELECT

Le **SQL** (Structured Query Language) est un langage permettant de communiquer avec une base de données. Ce langage informatique est notamment très utilisé par les développeurs web pour communiquer avec les données d'un site web.

L'utilisation la plus courante du langage SQL consiste à lire des données dans une base de données. Pour cela on utilise la commande « SELECT ».

# La commande SELECT

L'utilisation basique de cette commande s'effectue de la manière suivante :

```
SELECT nom_du_champ FROM nom_du_tableau
```

Cette requête SQL va **sélectionner** (SELECT) le champ "nom_du_champ" **provenant** (FROM) du tableau appelé "nom_du_tableau".
Il est aussi possible de sélectionner plusieurs champs.
Exemple :

```
SELECT nom, prenom, age FROM etudiants.
```

Reprenons l'exemple de la table Etudiants vu plus haut.
La requête **`SELECT nom, prenom, age FROM etudiants`** aurait ramené les données suivantes.

| Nom | Prenom | Age |
|---|---|---|
| Zidane | Zinedine | 44 |
| Perrin | Loic | 33 |
| Umtiti | Samuel | 24 |

### Obtenir toutes les colonnes d'un tableau

Il est possible de retourner automatiquement toutes les colonnes d'un tableau sans avoir à connaître le nom de toutes les colonnes. Au lieu de lister toutes les colonnes, il faut simplement utiliser le caractère "*" (étoile). C'est un joker qui permet de sélectionner toutes les colonnes. Il s'utilise de la manière suivante:

```
SELECT * FROM etudiants
```

Cette requête SQL retourne exactement les mêmes colonnes qu'il y a dans la base de données. Dans notre cas, le résultat sera donc:

| Id | Nom | Prenom | Age |
|---|---|---|---|
| 1 | Zidane | Zinedine | 44 |
| 2 | Perrin | Loic | 33 |
| 3 | Umtiti | Samuel | 24 |

## *SQL DISTINCT*

L'utilisation de la commande **SELECT** en SQL permet de lire toutes les données d'une ou plusieurs colonnes. Cette commande peut potentiellement afficher des lignes en doubles. Pour éviter des redondances dans les résultats il faut simplement ajouter **DISTINCT** après le mot **SELECT**.

L'utilisation basique de cette commande consiste alors à effectuer la requête suivante:

```
SELECT DISTINCT nom FROM etudiants
```

Cette requête sélectionne le champ “nom” de la table “etudiants” en évitant de retourner des doublons.

Par exemple si dans la table il y aurait 2 personnes dont le nom est Zidane, la requête aurait simplement retourné :

| Nom |
| --- |
| Zidane |

## *SQL WHERE*

La commande **WHERE** dans une requête SQL permet d’extraire les lignes d’une base de données qui respectent une condition. Cela permet d’obtenir uniquement les informations désirées.

### Syntaxe

La commande WHERE s’utilise en complément à une requête utilisant SELECT. La façon la plus simple de l’utiliser est la suivante:

```
SELECT nom_colonnes FROM nom_table WHERE condition
```

### Exemple

Imaginons une table appelée “client” qui contient le nom des clients, le nombre de commandes qu’ils ont effectués et leur ville:

| id | nom | nbr_commande | ville |
| --- | --- | --- | --- |
| **1** | Paul | 3 | paris |
| **2** | Maurice | 0 | rennes |
| **3** | Joséphine | 1 | toulouse |
| **4** | Gérard | 7 | paris |

Pour obtenir seulement la liste des clients qui habitent à Paris, il faut effectuer la requête suivante:

```
SELECT * FROM client WHERE ville = 'paris'
```

Cette requête retourne le résultat suivant:

| id | nom | nbr_commande | ville |
| --- | --- | --- | --- |
| **1** | Paul | 3 | paris |
| **4** | Gérard | 7 | paris |

**Attention**: dans notre cas tout est en minuscule donc il n’y a pas eu de problème. Cependant, si une table est sensible à la casse, il faut faire attention aux majuscules et minuscules.

### Opérateurs de comparaisons

Il existe plusieurs opérateurs de comparaisons. La liste ci-jointe présente quelques-uns des opérateurs les plus couramment utilisés.

| Opérateur | Description |
| --- | --- |
| = | Égale |

| | |
|---|---|
| **<>** | Pas égale |
| **!=** | Pas égale |
| **>** | Supérieur à |
| **<** | Inférieur à |
| **>=** | Supérieur ou égale à |
| **<=** | Inférieur ou égale à |
| **IN** | Liste de plusieurs valeurs possibles |
| **BETWEEN** | Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates) |
| **LIKE** | Recherche en spécifiant le début, milieu ou fin d'un mot. |
| **IS NULL** | Valeur est nulle |
| **IS NOT NULL** | Valeur n'est pas nulle |

**Attention : Il est possible de coupler plusieurs conditions ensemble grâce au opérateur AND (et) ou OR (ou) . Reprenons la table « Etudiant » vue plus haut .**
**Exemple :**

```
SELECT nom FROM etudiants WHERE nom ='Zidane' OR prenom
= 'Loic'
```

Ceci ramènera donc :

| Nom |
|---|
| Zidane |
| Perrin |

## *L'opérateur LIKE*

### Syntaxe

La syntaxe à utiliser pour utiliser l'opérateur LIKE est la suivante :

```
SELECT nom AS Nom_de_l_étudiant FROM etudiants WHERE nom LIKE
'%zid%'
```

Ceci ramènera donc :

| nom |
|---|
| **Zidane** |
| **Zidou** |
| **Azida** |

C'est-à-dire tous les noms qui contiennent « zid ».

D'autre exemple d'utilisation de l'opérateur LIKE

- LIKE '%a' : le caractère "%" est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaines de caractère qui se termine par un "a".
- LIKE 'a%' : ce modèle permet de rechercher toutes les lignes de "colonne" qui commence par un "a".
- LIKE '%a%' : ce modèle est utilisé pour rechercher tous les enregistrements qui utilisent le caractère "a".
- LIKE 'pa%on' : ce modèle permet de rechercher les chaines qui commence par "pa" et qui se terminent par "on", comme "pantalon" ou "pardon".
- LIKE 'a_c' : peu utilisé, le caractère "_" (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage "%" peut être remplacé par un nombre incalculable de caractères . Ainsi, ce modèle permet de retourner les lignes "aac", "abc" ou même "azc"

## *Les Alias*

Parfois les noms de colonnes peuvent ne pas avoir de nom explicite (exemple idJoueur ), c'est pourquoi il est possible de renommé la colonne grace au alias.

```
SELECT nom AS Nom_de_l_étudiant FROM etudiants WHERE nom
='Zidane' OR prenom = 'Loic'
```

Ceci ramènera donc :

| Nom_de_l_etudiant |
|---|
| **Zidane** |
| **Perrin** |

## *Les Calculs*

En SQL il est possible réaliser des calculs mathématique.
Exemple :

```
SELECT prix, prix/2 AS nouveau_prix FROM articles
```

Ceci ramènera donc :

| prix | Nouveau_prix |
|---|---|
| **24** | 12 |
| **20** | 10 |

Les opérateurs disponibles :

| Opérateur | Description |
|---|---|

| | |
|---|---|
| + | Addition |
| - | Soustraction |
| / | Division |

*Note : Attention à respecter les priorités de calcul comme en mathématique. Ex : (3*(5+6))*

## *Fonctions d'agrégation SQL*

Les fonctions d'agrégation dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble d'enregistrement. Étant données que ces fonctions s'appliquent à plusieurs lignes en même temps, elles permettent des opérations qui servent à récupérer l'enregistrement le plus petit, le plus grand ou bien encore de déterminer la valeur moyenne sur plusieurs enregistrement.

### Liste des fonctions d'agrégation statistiques

Les fonctions d'agrégation sont des fonctions idéales pour effectuer quelques statistiques de bases sur des tables. Les principales fonctions sont les suivantes :

- AVG() pour calculer la moyenne sur un ensemble d'enregistrement
- COUNT() pour compter le nombre d'enregistrement sur une table ou une colonne distincte
- MAX() pour récupérer la valeur maximum d'une colonne sur un ensemble de ligne. Cela s'applique à la fois pour des données numériques ou alphanumérique
- MIN() pour récupérer la valeur minimum de la même manière que MAX()
- SUM() pour calculer la somme sur un ensemble d'enregistrement

Exemple :

Soit la table Etudiants suivantes :

| nom | Age |
|---|---|
| **Zidane** | 42 |
| **Perrin** | 33 |

```
SELECT Count(*) AS nombre_etudiants FROM etudiants
```

**Ceci nous donnera**

| Nombre_etudiants |
|---|
| **2** |

```
SELECT Sum(Age) AS Total_age FROM etudiants WHERE nom
='Zidane' OR prenom = 'Loic'
```

Ceci nous donnera

| Total_age |
|---|
| **75** |

## *Le tri*

La commande ORDER BY permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.

### Syntaxe

Une requête où l'on souhaite filtrer l'ordre des résultats utilise la commande ORDER BY de la sorte :

```
SELECT colonne1, colonne2
FROM table
ORDER BY colonne1
```

Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le suffixe DESC après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une virgule. Une requête plus élaborée ressemblerait à cela :

```
SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC
```

Exemple :

Soit la table Etudiants suivantes :

| nom |
|---|
| **Zidane** |
| **Perrin** |
| **MBappe** |
| **Babar** |

```
SELECT * AS nombre_etudiants FROM etudiants
ORDER BY nom ASC
```

Ceci nous donnera

| nom |
|---|
| **Babar** |
| **MBappe** |
| **Perrin** |
| **Zidane** |

## *LE GROUP BY*

La commande GROUP BY est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat. Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.

### Syntaxe d'utilisation de GROUP BY

De façon générale, la commande GROUP BY s'utilise de la façon suivante

```
SELECT colonne1, fonction(colonne2)
FROM table
GROUP BY colonne1
```

**A noter** : cette commande doit toujours s'utiliser après la commande WHERE et avant la commande HAVING.

### Exemple d'utilisation

Prenons en considération une table "achat" qui résume les ventes d'une boutique :

| id | 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 |

Ce tableau contient une colonne qui sert d'identifiant pour chaque ligne, une autre qui contient le nom du client, le coût de la vente et la date d'achat.
Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

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

La fonction SUM() permet d'additionner la valeur de chaque tarif pour un même client. Le résultat sera donc le suivant :

| client | SUM(tarif) |
|---|---|
| **Pierre** | 262 |
| **Simon** | 47 |
| **Marie** | 38 |

## *SQL HAVING*

La condition HAVING en SQL est presque similaire à WHERE à la seule différence que HAVING permet de filtrer en utilisant des fonctions telles que SUM(), COUNT(), AVG(), MIN() ou MAX().

### Syntaxe

L'utilisation de HAVING s'utilise de la manière suivante :

```
SELECT colonne1, SUM(colonne2)
FROM nom_table
GROUP BY colonne1
HAVING fonction(colonne2) operateur valeur
```

Cela permet donc de **SÉLECTIONNER** les colonnes de la table “nom_table” en GROUPANT les lignes qui ont des valeurs identiques sur la colonne “colonne1” et que la condition de HAVING soit respectée.
**Important :** HAVING est très souvent utilisé en même temps que GROUP BY bien que ce ne soit pas obligatoire.

### Exemple

Pour utiliser un exemple concret, imaginons une table “achat” qui contient les achats de différents clients avec le coût du panier pour chaque achat.

| id | client | tarif | date_achat |
|---|---|---|---|
| **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 |

Si dans cette table on souhaite récupérer la liste des clients qui ont commandé plus de 40€, toutes commandes confondues alors il est possible d’utiliser la requête suivante :

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
HAVING SUM(tarif) > 40
```

**Résultat :**

| client | SUM(tarif) |
|---|---|
| **Pierre** | 262 |
| **Simon** | 47 |

## *SQL Sous-requête*

Dans le langage SQL une sous-requête (aussi appelé “requête imbriquée” ou “requête en cascade”) consiste à exécuter une requête à l’intérieur d’une autre requête. Une requête imbriquée est souvent utilisée au sein d’une clause WHERE ou de HAVING pou remplacer une ou plusieurs constante.

### Syntaxe

Il y a plusieurs façons d’utiliser les sous-requêtes. De cette façon il y a plusieurs syntaxes envisageables pour utiliser des requêtes dans des requêtes.

## Requête imbriquée qui retourne un seul résultat

L'exemple ci-dessous est un exemple typique d'une sous-requête qui retourne un seul résultat à la requête principale.

```
SELECT *
FROM `table`
WHERE `nom_colonne` = (
    SELECT TOP 1 `valeur`
    FROM `table2`

  )
```

Cet exemple montre une requête interne (celle sur "table2") qui renvoi une seule valeur. La requête externe quant à elle, va chercher les résultat de "table" et filtre les résultats à partir de la valeur retournée par la requête interne.

**A noter :** il est possible d'utiliser n'importe quel opérateur d'égalité tel que =, >, <, >=, <= ou <>.

## Requête imbriquée qui retourne une colonne

Une requête imbriquée peut également retournée une colonne entière. Dès lors, la requête externe peut utiliser la commande IN pour filtrer les lignes qui possèdent une des valeurs retournées par la requête interne. L'exemple ci-dessous met en évidence un tel cas de figure:

```
SELECT *
FROM `table`
WHERE `nom_colonne` IN (
    SELECT `colonne`
    FROM `table2`
    WHERE `cle_etrangere` = 36
  )
```

### Exemple

La suite de cet article présente des exemples concrets utilisant les sous-requêtes. Imaginons un site web qui permet de poser des questions et d'y répondre. Un tel site possède une base de données avec une table pour les questions et une autre pour les réponses.

**Table "question" :**

| q_id | q_date_ajout | q_titre | q_contenu |
|---|---|---|---|
| **1** | 2013-03-24 12:54:32 | Comment réparer un ordinateur? | Bonjour, j'ai mon ordinateur de cassé, comment puis-je procéder pour le réparer? |
| **2** | 2013-03-26 19:27:41 | Comment changer un pneu? | Quel est la meilleur méthode pour changer un pneu facilement ? |
| **3** | 2013-04-18 20:09:56 | Que faire si un appareil est cassé? | Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux? |
| **4** | 2013-04-22 17:14:27 | Comment faire nettoyer un clavier d'ordinateur? | Bonjour, sous mon clavier d'ordinateur il y a beaucoup de poussière, comment faut-il procéder pour le nettoyer? Merci. |

**Table "reponse" :**

| r_id | r_fk_question_id | r_date_ajout | r_contenu |
|---|---|---|---|
| **1** | 1 | 2013-03-27 07:44:32 | Bonjour. Pouvez-vous expliquer ce qui ne fonctionne pas avec votre ordinateur? Merci. |
| **2** | 1 | 2013-03-28 19:27:11 | Bonsoir, le plus simple consiste à faire appel à un professionnel pour réparer un ordinateur. Cordialement, |
| **3** | 2 | 2013-05-09 22:10:09 | Des conseils son disponible sur internet sur ce sujet. |
| **4** | 3 | 2013-05-24 09:47:12 | Bonjour. Ça dépend de vous, de votre budget et de vos préférence vis-à-vis de l'écologie. Cordialement, |

## Requête imbriquée qui retourne un seul résultat

Avec une telle application, il est peut-être utile de connaître la question liée à la dernière réponse ajoutée sur l'application. Cela peut être effectué via la requête SQL suivante:

```
SELECT *
FROM `question`
WHERE q_id = (
    SELECT TOP 1 r_fk_question_id
    FROM `reponse`
    ORDER BY r_date_ajout DESC  )
```

Une telle requête va retourner la ligne suivante:

**Note : Le TOP 1 retourne la première ligne correspondante à la requête.**

| q_id | q_date_ajout | q_titre | q_contenu |
|---|---|---|---|
| **3** | 2013-04-18 20:09:56 | Que faire si un appareil est cassé? | Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux? |

Ce résultat démontre que la question liée à la dernière réponse sur le forum est bien trouvée à partir de ce résultat.

## Requête imbriquée qui retourne une colonne

Imaginons maintenant que l'on souhaite obtenir les questions liées à toutes les réponses comprises entre 2 dates. Ces questions peuvent être récupérées par la requête SQL suivante:

```
SELECT *
FROM `question`
WHERE q_id IN (
    SELECT r_fk_question_id
    FROM `reponse`
    WHERE r_date_ajout BETWEEN '2013-01-01' AND '2013-12-31'
  )
```

Dans notre cas, cette requête retournera les résultats suivants :

| q_id | q_date_ajout | q_titre | q_contenu |
|---|---|---|---|
| **1** | 2013-03-24 12:54:32 | Comment réparer un ordinateur? | Bonjour, j'ai mon ordinateur de cassé, comment puis-je procéder pour le réparer? |

| | | | |
|---|---|---|---|
| **2** | 2013-03-26 19:27:41 | Comment changer un pneu? | Quel est la meilleur méthode pour changer un pneu facilement ? |
| **3** | 2013-04-18 20:09:56 | Que faire si un appareil est cassé? | Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux? |

Une telle requête permet donc de récupérer les questions qui ont eu des réponses entre 2 dates. C'est pratique dans notre cas pour éviter d'obtenir des réponses qui n'ont pas eu de réponses du tout ou pas de nouvelles réponses depuis longtemps.

## *Jointure SQL*

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

### Exemple

En général, les jointures consistent à associer des lignes de 2 tables en associant l'égalité des valeurs d'une colonne d'une première table par rapport à la valeur d'une colonne d'une seconde table. Imaginons qu'une base de 2 données possède une table "utilisateur" et une autre table "adresse" qui contient les adresses de ces utilisateurs. Avec une jointure, il est possible d'obtenir les données de l'utilisateur et de son adresse en une seule requête.
On peut aussi imaginer qu'un site web possède une table pour les articles (titre, contenu, date de publication …) et une autre pour les rédacteurs (nom, date d'inscription, date de naissance …). Avec une jointure il est possible d'effectuer une seule recherche pour afficher un article et le nom du rédacteur. Cela évite d'avoir à afficher le nom du rédacteur dans la table "article".
Il y a d'autres cas de jointures, incluant des jointures sur la même table ou des jointure d'inégalité. Ces cas étant assez particulier et pas si simple à comprendre, ils ne seront pas élaboré sur cette page.

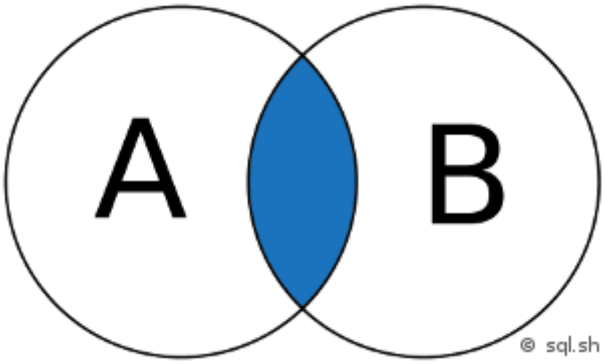
### Types de jointures

Il y a plusieurs méthodes pour associer 2 tables ensemble. Voici les trois types de jointures les plus utilisées

- **INNER JOIN :** jointure interne pour retourner les enregistrements quand la condition est vraie dans les 2 tables. C'est l'une des jointures les plus communes.
- **LEFT JOIN (ou LEFT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifié dans l'autre table.
- **RIGHT JOIN (ou RIGHT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifié dans l'autre table.

### Exemples de jointures

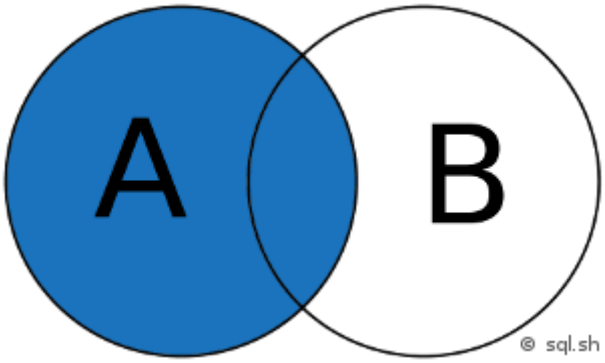
## INNER JOIN




Intersection de 2 ensembles

```
SELECT *
FROM A
INNER JOIN B ON A.key = B.key
```

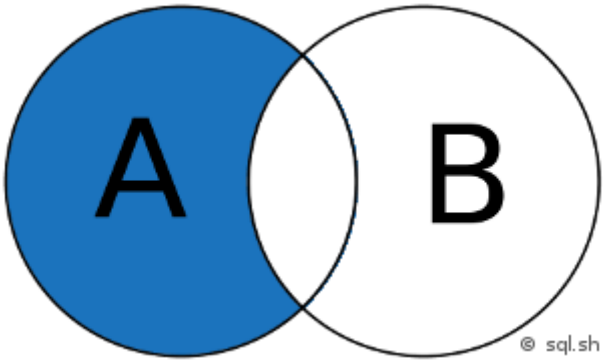
## LEFT JOIN




Jointure gauche (LEFT JOIN)

```
SELECT *
FROM A
LEFT JOIN B ON A.key = B.key
```

## LEFT JOIN (sans l'intersection de B)

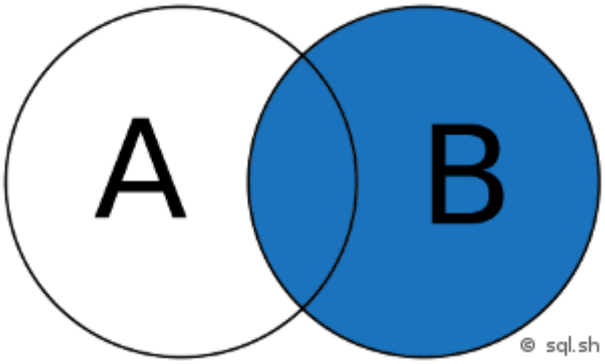



Jointure gauche (LEFT JOIN sans l'intersection B)

```
SELECT *
FROM A
```

```
LEFT JOIN B ON A.key = B.key
WHERE B.key IS NULL
```

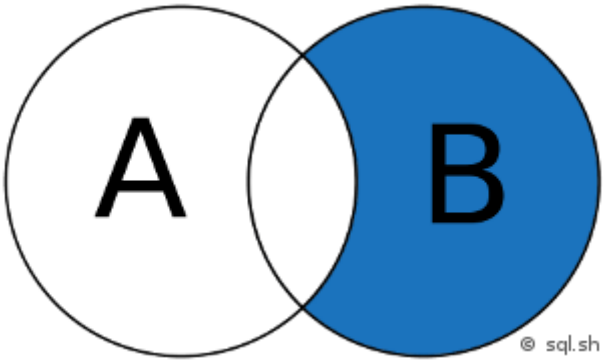
## RIGHT JOIN




Jointure droite (RIGHT JOIN)

```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
```

## RIGHT JOIN (sans l'intersection de A)




Jointure droite (RIGHT JOIN sans l'intersection A)

```
SELECT *
FROM A
RIGHT JOIN B ON A.key = B.key
WHERE B.key IS NULL
```

Source : Moi et le site
https://sql.sh/

Pour plus de détails vous pouvez trouver un cours détailler sur :
https://sql.sh/ressources/cours-sql-sh-.pdf