

Bases de données

Synthèses pour le cours théorique de base de donnée ainsi que Oracle SQL et SQL server

- [Introduction](#)
- [Architecture ANSI/SPARC et modélisation](#)
- [Modèle relationnel](#)
- [Oracle SQL](#)
- [Gérer des tables](#)
- [Gérer des tuples](#)
- [Les jointures](#)
- [Les fonctions spéciales](#)
- [Les groupements, requetes imbriquées et opérateurs ensemblistes](#)
- [Les vues et contraintes](#)
- [Triggers](#)

Introduction

Une base de donnée est un système qui permet de mémoriser de manière durable des données sur un support physique (mémoire secondaire)

Les données devront pouvoir être créées, lues, modifiées, et supprimées (CRUD, create read update delete).

Les objets représentés dans la base de donnée pourront être organisés de manière à faire apparaître les relations entre elles.

Tous les informaticiens sont confrontés un jour ou l'autre à une base de donnée. Mais il existe aussi un profil spécifique dédié à l'administration de bases de données (DBA) qui est responsable du bon fonctionnement des serveurs de bases de données et de leur création.

Qu'est ce qu'un SGBD

Un SGBD est un *Système de Gestion de Bases de Données* dans le but d'optimiser les performances, de protéger l'intégrité des données, même en cas de panne, de seulement autoriser l'accès aux personnes autorisées et de partager des données.

Un SGBD propose aussi des outils divers pour aider à la conception et à la gestion des bases de données.

Modèle de SGBD

Une base de donnée relationnelle est organisée en tables. Chaque table contenant des colonnes et des lignes.

Le modèle hiérarchique classe les données hiérarchiquement selon une arborescence descendante (premier modèle)

Le modèle réseau, une extension du modèle hiérarchique selon une structure de graphe

Le modèle orienté-objet,

Le modèle spatial

Le modèle XML

NOSQL, qui permet de palier aux limites du modèle relationnel, les données peuvent être mises horizontalement à travers plusieurs instances pour permettre à un serveur à ne pas avoir à gérer tout.

Architecture ANSI/SPARC et modélisation

Architecture ANSI/SPARC

L'architecture ANSI/SPARC est organisée en 3 niveaux:

- Le niveau interne (physique) qui définit la structure du stockage sur le support (fichiers, secondaire). Ce niveau interesse principalement les développeurs de SGBD et les DBA.
- Le niveau conceptuel (décrit l'organisation de l'ensemble des données, et leur contraintes). Ce niveau interesse principalement les développeurs.
- Le niveau externe propose aux groupes utilisateurs sa propre vue des données dont il a besoin.

Description des données et création de modèles

Pour créer un schéma de base de données il faut :

- Déterminer comment représenter les objets (caractéristiques, comment l'identifier, etc)
- Déterminer les liens qui unissent les objets (ainsi que les contraintes sur ces liens)
- Déterminer les contraintes sur les caractéristiques (types, maximum, minimum, longueur, etc)

Généralement ça commence par la création d'un "schéma entité-association" (modèle conceptuel des données) ensuite converti dans un modèle logique, puis complété par des éléments techniques tel que les types et les contraintes. Enfin les éléments du schéma sont traduits en instructions de création de la bdd.

- Le niveau conceptuel ignore les aspects propre à l'informatique. Se concentre sur le métier de l'organisme et débouche sur un *MCD* (entité-association)
- Le niveau logique (dépend du type de SGBD utilisé) permet d'ajouter des contraintes sur les données (types ou autres) et débouche sur un *MLD* (Modèle Logique des Données)
- Le niveau physique se concentre sur les spécificités du SGBD utilisé, ajout des types spécifiques, et création du script SQL pour la création des tables.

Modèle relationnel

Domaine (théorique)

Le domaine est l'ensemble des valeurs possible dans une base de donnée. Par exemple:

- Ensemble des entiers : $\{1, 2, 3, 4, 5, \dots\}$
- Ensemble des entiers allant de 1 à 4 : $\{1, 2, 3, 4\}$
- Ensemble des noms : $\{\text{Robert, Jean, Thomas, Marc, Jules, Simone}\}$
- ...

En pratique, ce domaine est défini par les types et les contraintes.

Produit cartésien (théorique)

Le produit cartésien est la "mise en relation" de plusieurs domaines. Par exemple :

Produit cartésien des noms et des entiers allant de 1 à 4 donne 24 combinaisons possibles. Mais seul 6 seront utilisée (une par nom).

Une relation

Une relation est un sous-ensemble nommé d'un *produit cartésien*. Dans l'exemple précédent, cela donnera par exemple : (toutes les occurences sont des Personnes qui sont donc dans la table "Personnes")

Nom	Age
Robert	2
Jean	1
Thomas	2
Marc	4
Jules	3
Simone	1

Une relation est représentée par une table. Une table (ou relation) est consituée de :

- Une colonne représente un attribut, une propriété. Par exemple: "Age"

- Une ligne (tuple) représente une occurrence et est unique. Par exemple "Simone - 1" est un tuple

La cardinalité d'une relation est le nombre de tuples qui la compose, et le degré de la relation est le nombre d'attributs.

Schéma d'une relation

Le schéma d'une relation (ou d'une table) consiste de :

- Du nom de la relation, exemple, « *Personnes* »
- De la liste des attributs. Pour chaque attribut est définis:
 - Un nom, exemple, « *Nom* »
 - Un type, exemple, une chaîne de caractère de 25 caractères maximum « *varchar(25)* »
 - Potentiellement des contraintes, par exemple, ne peut pas être nulle « *NOT NULL* »

En SQL voici un exemple de schéma:

```
CREATE TABLE Personnes (  
  id INT NOT NULL,  
  nom VARCHAR(25) NOT NULL,  
  age INT NOT NULL,  
  PRIMARY KEY (id)  
);
```

Pour juger de la qualité d'un schéma il faut s'assurer que le schéma :

- Ne va pas créer de redondances
- Ne va pas créer d'anomalies
 - Anomalie de mise à jour, une mise à jour d'une information n'est pas répercutée sur toutes les occurrences de l'information
 - Anomalie de suppression, la suppression d'un tuple entraîne une perte d'information pertinente

Types de clés

- Les clés **candidates** est un sous-ensemble d'attributs tel que
 - Doivent être uniques (contrainte d'unicité), 2 tuples ne peuvent pas avoir la même valeur pour cet ensemble d'attributs
 - Les attributs présents doivent être absolument nécessaires (contrainte d'irréductibilité)

Exemples de clé candidates : Matricule, nom + prénom + adresse, etc.

- La clé **primaire** est un attribut choisi parmi les clés candidates choisi pour identifier un tuple. Dans la représentation une clé primaire sera soulignée.

Exemple de clé primaire : Matricule

- Les clés **étrangères** sont des références à des clés candidates (souvent la clé primaire), d'une autre table ou de la même table, soit de faire des liens entre des tuples.
 - Elle doivent faire référence à une clé primaire qui existe déjà
 - Si une clé étrangère existe pour une clé primaire on ne peut supprimer la clé primaire sans supprimer la clé étrangère. (ces deux règles sont la contrainte d'intégrité référentielle)

Exemple de clé étrangère : Mention d'un matricule dans une autre table

- Les **surclés** sont des ensembles qui ne respectent pas la contrainte d'irréductibilité des clés candidates. Soit une clé candidates (+ d'autres attributs) = surclés.

Exemples de surclé : matricule, matricule + nom

Les contraintes

- La contrainte d'unicité (**UNIQUE**) signifie que qu'il ne peut pas y avoir 2 fois la même valeur pour cet attribut dans la table
- La contrainte **NOT NULL** signifie qu'il faut y avoir une valeur pour cet attribut, il signifie que l'attribut est obligatoire

Types primitifs des attributs

Les types peuvent être différents d'un SGBD à un autre. Par exemple dans Oracle, le type **boolean** n'existe pas il faut donc créer une colonne **integer** qui a un domaine de $\{0,1\}$.

- Type **integer**, seulement les nombres entiers
- Type **numeric(precision[, scale])** admet des nombres réels, **precision** indique le nombre de chiffre et **scale** indique le nombre de chiffres décimaux (après la virgule)
- Type **char(longueur)** indique une chaîne de caractère qui prends un espace fixe dans la base de donnée.
- Type **varchar(longueur)** indique la même chose que le précédent, sauf que l'espace est variable
- Type **date** indique une date (jour, mois, année)
- Type **timestamp** indique une date et une heure
- Type **boolean** admet les valeurs "vrai" ou "faux" (1 ou 0)

Les opérateurs relationnels (algèbre relationnelle)

- La sélection (`SELECT ... WHERE ...`) qui permet d'obtenir une nouvelle relation de même schéma qui satisfairont une condition donnée.
- La projection (`SELECT ...`) permet d'obtenir, par une sélection, seulement certains attributs.
- L'union de deux relations permet de combiner deux tables (ayant le même schéma) en une seule liste de tuple.
- L'intersection est comme l'union, elle combine deux table, sauf que contrairement à l'union, elle supprime les doublons de tuples.
- La jointure

La normalisation (les formes normales)

En base de données relationnelles on défini différentes formes normales, elles permettent d'éviter la redondance des données et d'utiliser le SGBD à son plein potentiel.

La première forme normale (1FN) défini qu'il ne peut pas y avoir des tuples en doublon dans une table et que les attributs ne doivent pas être décomposable (par exemple "nom_prenom" est décomposable en "nom" et "prenom"), on dit alors que les attributs sont "atomiques"

La deuxième forme normale (2FN) défini qu'aucun attribut ne peut dépendre d'uniquement une partie de la clé primaire. Par exemple:

Employé
nom
prenom
nom_département
description_département

Si on imagine une table "employé" qui est identifié par la clé primaire "nom + prenom + nom_département". On peut voir que "description_département" dépend d'une partie de la clé car il est directement lié à "nom_département". Par conséquent cela n'est pas une 2e forme normale.

Pour que celle ci soit en une deuxième forme normale on peut soit, définir un seul attribut comme clé primaire (exemple, "id_employé") ou mettre description_département dans une autre table et utiliser nom_département comme clé primaire de celle ci:

Employé
nom
prenom
nom_departement

Departement
nom_departement
description_departement

Pour qu'une table soit en 3e forme normale (3FN) il faut qu'aucun attribut ne dépende d'autres attributs (autre que la clé primaire), imaginons l'exemple précédent sauf que **nom_departement** ne fait plus partie de la clé primaire :

Employé
nom
prenom
nom_departement
description_departement

Ici description_departement dépend de nom_departement, par conséquent cette table n'est pas en 3e forme normale. On peut séparer de nouveau en deux tables comme vu précédemment pour régler ce problème.

Pour résumer les formes normales, pour tester une table on doit se poser ces questions :

- Les attributs sont-ils atomiques (non décomposables) ? Si oui -> 1FN
- Si 1FN, est ce qu'aucun attribut ne dépend d'une partie de la clé primaire / il n'y qu'un seul attribut comme clé primaire ? Si oui -> 2FN
- Si 2FN, est ce qu'aucun attribut ne dépend d'autres attribut que la clé primaire ? Si oui -> 3FN

Oracle SQL

SQL veut dire Structured Query Language et permet d'interagir avec les bases de données. Les SGBD étant différents, les langages SQL varient aussi même si certaines choses sont standardisées.

SQL est divisé en 4 parties

- DDL (Data Definition Language) qui permet de gérer le schéma d'une base de données
- DML (Data Manipulation Language) qui permet de gérer les données en elle-même (les ajouter, supprimer, modifier et lire)
- DCL (Data Control Language) qui permet de gérer les permissions aux données pour chaque utilisateur SGBD. Ces utilisateurs ne correspondent pas aux utilisateurs finaux mais bien aux utilisateurs du SGBD (donc seulement sur le serveur)

Attention: Tout ce chapitre est consacré à Oracle SQL. Et beaucoup de choses peuvent différer beaucoup, voire ne pas exister dans d'autres SGBD.

Gérer des tables

Dans cet exemple on crée une table "Persons" qui a les colonnes (attributs) PersonID, LastName, FirstName, Address et City qui ont toutes des types particuliers tel que `varchar(255)` qui indique une chaîne de caractère de taille variable et de maximum 255 caractères, et `int` pour un entier.

```
CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(255),  
    FirstName varchar(255),  
    Address varchar(255),  
    City varchar(255)  
);
```

En utilisant `ALTER TABLE` on peut modifier les informations d'une table que l'on a déjà créé. Dans cet exemple on ajoute une contrainte `PRIMARY KEY` pour la colonne `PersonID`. Ceci est la même chose que d'ajouter `PRIMARY KEY (ID)` à la fin de la requête de création de table.

```
ALTER TABLE Persons ADD PRIMARY KEY (PersonID);
```

Imaginons que nous n'avons plus besoin de cette table, on peut alors la supprimer avec `DROP`.

```
DROP TABLE Persons;
```

Gérer des tuples

Dans cet exemple on ajoute une nouvelle ligne (tuple) dans notre table, on précise d'abord la liste des colonnes auquel on va ajouter un résultat, puis on ajoute les valeurs correspondantes.

```
INSERT INTO Persons (PersonID, LastName, FirstName, Address, City) VALUES (1, 'Shumi', 'Roger', 'Rue du moulin, 13', 'Liège');
```

Les deux lignes décrites ci-dessus ont le même effet. Elles sélectionnent tous les tuples qui correspondent à la condition "PersonID=1" de la table "Persons" avec toutes les propriétés (c'est ce que veut aussi dire le `*`, c'est un "wildcard" qui inclut toutes les propriétés).

```
SELECT (PersonID, LastName, FirstName, Address, City) FROM Persons WHERE PersonID=1;  
SELECT * FROM Persons WHERE PersonID=1;
```

Dans cet exemple on change les attributs Address et City dans tous les tuples qui remplissent la condition `PersonsID=1` dans la table "Persons". A noter que quand on a une chaîne de caractères qui contient un `'` on peut quand même l'inclure en doublant le `'`.

```
UPDATE Persons SET Address='Rue de l''église', City='Verviers' WHERE PersonsID=1;
```

Mais on peut aussi modifier une table en se basant sur les valeurs de celle-ci.

```
-- Ceci est un exemple dans une autre table où l'on modifie le montant en l'augmentant de 10%  
UPDATE Bien SET montant = montant * 1.1 WHERE id_bien = 4;
```

On supprime Roger en supprimant tous les éléments de notre table "Persons" où `PersonsID=1`.

```
DELETE FROM Persons WHERE PersonsID=1;
```

Des SELECT plus avancés

Maintenant pour avoir un meilleur formatage, on peut utiliser des clauses `SELECT` plus évoluées.

Voici un exemple avec une date que l'on formate en texte (sous le format tel que '12/12/2022') :

```
SELECT TO_CHAR(date_offre, 'dd/mm/yy')  
FROM Offre
```

Renommer le nom d'une colonne avec `AS`

```
SELECT TO_CHAR(date_offre, 'dd/mm/yy') AS date
FROM Offre
```

Concaténer des colonnes et des chaînes de caractères

```
SELECT 'La date est ' || date_offre || ' et le statut est ' || statut AS ma_super_colonne_inutile
FROM Offre
```

Enfin si on veut éviter que des lignes apparaissent plusieurs fois on peut utiliser le mot-clé `DISTINCT`

```
-- On prends les localités de tous les candidats mais on supprime toutes les lignes en doublon
SELECT DISTINCT localite
FROM Candidat
```

Des WHERE plus avancés

On peut faire différentes opérations de base dans les clauses `WHERE` tel que `>`, `<`, `<=`, `>=`, `=`

```
SELECT * FROM Offre WHERE id_offre = 4
```

On peut ensuite lier plusieurs comparaisons avec des opérateurs booléens (AND, OR, NOT)

```
SELECT * FROM Offre WHERE NOT id_offre = 4 AND montant > 1000;
```

Pour vérifier si il n'y a aucune valeur, on peut utiliser `IS NULL`

```
SELECT * FROM Offre WHERE id_candidat IS NULL;
```

Enfin pour tester des patterns de chaînes de caractères on peut utiliser le mot clé `LIKE`

```
-- Ne retourne que les tuples de la table bien qui ont 'calme' en lowercase dans leur description
-- % signifie "n'importe quel chaîne de caractère"
-- _ signifie "n'importe quel caractère"
SELECT * FROM Bien WHERE description LIKE '%calme%';

-- Ici on veut tous les tuples de Bien qui ont le symbole % dans leur description,
-- mais % est un caractère de syntaxe de LIKE donc on doit l'escape ici on choisit \
```

```
-- Si on veut après escape '\' alors on peut simplement le mettre deux fois '\\'
SELECT * FROM Bien WHERE description LIKE '%\\%' ESCAPE '\\';
```

Pour vérifier si une valeur est présente dans une certaine liste de valeur il est préférable d'utiliser **IN**

```
-- Ce code va garder tous les tuples dont la localité est soit Liège, soit Neupré
SELECT id_bien, localite
FROM Bien
WHERE localite IN ('Liège', 'Neupré');
```

Maintenant si on veut facilement savoir si une valeur est entre 2 valeurs on peut utiliser le mot-clé **BETWEEN**

```
-- Le mot-clé BETWEEN fonctionne aussi avec des dates par exemple
SELECT id_offre, date_offre, montant
FROM Offre
WHERE montant BETWEEN 200000 AND 300000
```

Utiliser ORDER BY pour changer l'ordre des tuples

On peut choisir comment on veut organiser nos tuples (ordre ascendant, descendant, alphabétique, numérique, etc)

```
-- Ceci va faire apparaitre toutes les Offres ordonné par le montant dans un ordre descendant (du plus cher au moins cher)
-- Si on remplace DESC par ASC, on aura l'inverse (du moins cher au plus cher)
SELECT * FROM Offre
ORDER BY montant DESC;
-- OUTPUT:  ID_OFFRE  MONTANT DATE_OFF S ID_CANDIDAT  ID_BIEN
--  -----
--      19   262000 13/10/12 E      19      19
--      11   260000 02/10/12 I      11      19
--       7   257500 17/09/12 R       7      19

-- On peut aussi mettre plusieurs ORDER BY à la suite comme 'ORDER BY montant DESC, id_offre ASC'
-- En faisant ça tous les tuples qui ont le même montant seront ordonnés par l'id_offre
```


Les jointures

Diagramme fort utile sur les jointures

On définit donc quel est notre table "de gauche" et notre table "de droite". Ensuite dépendant des informations que l'on veut obtenir on va utiliser différents types de jointures.

- Pour avoir seulement les tuples ayant une certaine valeur commune dans les 2 tables

```
SELECT b.id_bien, b.localite, c.prenom, c.nom
FROM Bien b
JOIN Client c ON b.id_client = c.id_client
```

- Pour avoir toute la table de gauche + les tuples communs aux deux tables

```
SELECT c.nom, c.prenom, o.montant, o.statut
FROM Candidat c
LEFT OUTER JOIN Offre o ON o.id_candidat = c.id_candidat
```

- Pour avoir **uniquement** les tuples unique à la table de gauche, et non présent sur la table de droite

```
SELECT c.nom, c.prenom, o.montant, o.statut
FROM Candidat c
LEFT OUTER JOIN Offre o ON o.id_candidat = c.id_candidat
WHERE o.id_candidat IS NULL
```

- Pour avoir toutes les données des deux tables

```
SELECT c.nom, c.prenom, o.montant, o.statut
FROM Candidat c
FULL OUTER JOIN Offre o ON c.id_candidat = o.id_candidat
```

- Pour avoir seulement les tuples unique à chaque table, mais pas ceux qui sont communs

```
SELECT c.nom, c.prenom, o.montant, o.statut
FROM Candidat c
FULL OUTER JOIN Offre o ON c.id_candidat = o.id_candidat
WHERE c.id_candidat IS NULL OR o.id_candidat IS NULL
```


- Pour tous les `RIGHT OUTER JOIN` c'est la même chose que dit plus tot, mais en inversant comme vu dans le schéma plus haut

Quand il y a plusieurs joins (soit plus de 2 tables)

Après un JOIN, les deux tables liées se comporte comme une nouvelle table. Soit du point de vue du JOIN suivant, le résultat du premier JOIN se comporte comme si c'était la table 'A' sur le schéma.

Voici un exemple provenant de l'examen formatif de décembre : [Énoncé de la question](#)

Donc si on représente la situation avec des diagrammes ça nous donne ceci :

Diagramme de la situation

Et une fois représenté en SQL ça nous donne :

```
-- On prends les valeurs que l'on veut dans le select
SELECT v.nom AS nom_vendeur, v.prenom AS prenom_vendeur, m.texte AS message,
TO_CHAR(m.date_message, 'dd-month-yyyy') AS date_message
-- On va partir de la table Message (mais on pourrait aussi partir de vendeur)
FROM Message m
-- On va regrouper Message et Alerter
LEFT OUTER JOIN Alerter a ON a.id_message = m.id_message
-- On va ajouter Vendeur à notre (Message + Alerter)
FULL OUTER JOIN Vendeur v ON a.id_vendeur = v.id_vendeur
-- On va éliminer tout les tuples qui sont reliés en supprimant tous les liens de Alerter
WHERE a.id_message IS NULL;
```

Les fonctions spéciales

Il y a plusieurs fonctions spéciales en SQL qui peuvent être ajoutée plus ou moins n'importe où dans la requête.

Conversion des dates

Pour convertir un `DATE` en chaîne de caractère, on peut utiliser `TO_CHAR()`

```
SELECT localite, TO_CHAR(date_mise_en_vente, 'month-yyyy') FROM Bien;
```

Pour ce qui est de la syntaxe voici un extrait :

Syntaxe SQL	Correspond à	Exemple
<code>dd</code>	Jour du mois	<code>05</code>
<code>mm</code>	Mois en numéro	<code>02</code>
<code>yyyy</code>	Année en 4 numéros	<code>2022</code>
<code>yy</code>	Année en 2 numéros	<code>22</code>
<code>month</code>	Mois (épilé)	<code>décembre</code>
<code>mon</code>	Nom du mois en abrégé	<code>JAN</code>
<code>ddd</code>	Jour de l'année	<code>365</code>
<code>day</code>	Le jour en lettres	<code>lundi</code>
<code>mi</code>	Minutes	<code>56</code>
<code>hh</code>	Heures en format 12h	<code>1</code>
<code>ss</code>	Secondes	<code>59</code>
<code>hh24</code>	Heures en format 24h	<code>13</code>
<code>am</code>	Afficher AM si matin ou PM si après midi	<code>PM</code>

Pour la liste complète du format de `TO_CHAR` [cliquez ici](#)

Le même principe peut être utilisé pour convertir des chaînes de caractères en date aussi avec `TO_DATE()` qui a la même syntaxe.

- Si on veut juste extraire sans formater certaines valeurs de `DATE` ou `TIME` on peut utiliser la fonction `EXTRACT()`

```
-- YEAR peut être remplacé par MONTH, DAY. Et dans les cas où c'est `TIME` également par HOUR, MINUTE ou SECOND
SELECT EXTRACT(YEAR FROM date_mise_en_vente) AS annee FROM Bien;
-- OUTPUT: 2012
```

- On peut aussi récupérer la date actuelle du système avec la fonction `SYSDATE()`

```
-- On ajoute dans une table 'Leaderboard' la date avec SYSDATE
INSERT INTO Leaderboard (date, nom, prenom, score) VALUES (SYSDATE, 'Roger', 'Pierre', 144);
```

- On peut arrondir une date à une certaine unité avec `ROUND()`

```
-- On va arrondir la date au mois ce qui va retourner des choses tel que '01-12-2022' à la place de '13-12-2022'
SELECT ROUND(date_mise_en_vente, 'month') FROM Bien;
-- \_ Les valeurs ici sont les mêmes que celle vue dans le tableau au dessus
```

- On peut ajouter un certain nombre de mois à une date avec `ADD_MONTHS()`

```
-- Affiche la date d'aujourd'hui, la date d'aujourd'hui + 5 mois et le nombre de jours entre les 2
SELECT SYSDATE AS aujourd'hui,
       ADD_MONTHS(SYSDATE, 5) AS deadline,
       ADD_MONTHS(SYSDATE, 5) - SYSDATE AS tempsrestant
FROM Bien;
-- OUTPUT: AUJOURDH DEADLINE JOURSRENTANTS
-- -----
-- 30/12/22 30/05/23      151
```

- Affiche le dernier jour du mois avec `LAST_DAY()`

```
SELECT SYSDATE AS aujourd'hui, LAST_DAY(SYSDATE) AS findumois
FROM Bien;
-- OUTPUT: AUJOURDH FINDUMOI
-- -----
-- 30/12/22 31/12/22
```

- Affiche le nombre de mois de différence entre deux dates avec `MONTHS_BETWEEN()`

```
-- Ici je vais reprendre un exemple précédent mais changer le 'tempsrestant'
SELECT SYSDATE AS aujourd'hui,
       ADD_MONTHS(SYSDATE, 5) AS deadline,
       MONTHS_BETWEEN(ADD_MONTHS(SYSDATE, 5), SYSDATE) AS tempsrestant
FROM Bien;
-- OUTPUT: AUJOURDH DEADLINE TEMPSRESTANT
--
--      -----
--      30/12/22 30/05/23      5
```

- Pour avoir la date du premier jour de la semaine donné à partir d'une date on peut utiliser `NEXT_DAY()`

```
-- Donne la date du prochain lundi à partir d'aujourd'hui
SELECT SYSDATE AS aujourd'hui,
       NEXT_DAY(SYSDATE, 'LUNDI') AS prochainlundi
FROM Bien;
-- OUTPUT: AUJOURDH PROCHAIN
--
--      -----
--      30/12/22 02/01/23
```

Conversion des chaines de caractères

On peut utiliser `LOWER()` pour mettre en minuscule, ou `UPPER()` pour mettre en majuscule.

```
-- Voici un exemple de SQL avec un LIKE qui est case-insensitive en le transformant en lowercase
SELECT * FROM Bien WHERE LOWER(description) = '%calme%';
```

Enfin on peut aussi utiliser `SUBSTR()` pour couper une chaîne de caractère (c'est très similaire au *substring* en Java)

```
-- Va donner l'initiale du prenom + le nom des candidats et renommer la colonne 'candidat'
-- On donne ici la position de début (1) et la position de fin (1) dans la chaîne.
-- Mais si on veut prendre tout à partir de la position 2 par exemple, on peut juste écrire SUBSTR(nom, 2)
SELECT SUBSTR(prenom, 1, 1) || ' ' || nom AS candidat FROM Candidat;
```

On peut utiliser la fonction `replace` pour remplacer une chaîne par une autre

```
-- Remplace tous les "robert" par des "roger"
SELECT REPLACE(prenom, 'Robert', 'Roger') AS prenom FROM candidat;
```

Mais dans certains cas on veut éliminer toute une série de caractère de nos entrées (pour pouvoir les utiliser comme nom de fichier par exemple) en utilisant `TRANSLATE`

```
-- Remplace les caractères espace, apostrophe, point, slash, backslash et étoile par des underscore
SELECT TRANSLATE(nom, ' ""'.\*', ' _____') AS filename FROM fichiers;
```

Pour savoir la longueur d'une chaîne de caractère on peut utiliser la fonction `LENGTH`

```
-- Donne pour chaque candidat, son nom et la longueur de son nom
SELECT nom, LENGTH(nom) AS longueurdunom FROM candidat;
```

Pour savoir la position d'une chaîne de caractère dans une autre chaîne, on peut utiliser `INSTR`

```
-- L'exemple ici va donner 14.
-- Il va rechercher la 2e occurrence de OR dans la chaîne à partir du 3e caractère
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) FROM Bien;

--      |          |  |  \_Quelle occurrence
--      |          |  \_A partir de X caractère
--      \_Input      \_Chaîne à rechercher
```

Pour simplement retirer des caractères, sans utiliser `REPLACE` on peut utiliser `TRIM` et ses variantes

```
-- Pour remplacer le début d'une chaîne de caractères
SELECT LTRIM('foo bar', 'foo ') FROM Bien;
-- OUTPUT: 'bar'

-- Pour remplacer la fin d'une chaîne de caractères
SELECT RTRIM('foo bar', ' bar') FROM Bien;
-- OUTPUT: 'foo'

-- Pour remplacer le début et la fin d'une chaîne de caractères
-- ATTENTION: TRIM ne fonctionne que pour 1 caractère
SELECT TRIM(BOTH 'o' FROM 'oobaroo') FROM Bien;
-- OUTPUT: 'bar'

-- Note: 'BOTH' peut être remplacé par LEADING pour isoler le début ou par TRAILING pour isoler la fin
```

On peut convertir une chaîne de caractère en nombre avec `TO_NUMBER`

```
SELECT TO_NUMBER('1210.73', '9999.99') FROM Bien;
```

Gestion des nombres

- La fonction `ROUND()` fonctionne également pour les nombres

```
-- Ceci donne le nombre de jour qui sépare la date de mise en vente de chaque bien de la date d'aujourd'hui
-- Ce nombre est normalement un nombre à virgule mais on l'arrondi avec ROUND
SELECT ROUND(SYSDATE - date_mise_en_vente) AS nbr_jours
FROM BIEN;
-- OUTPUT: NBR_JOURS
--
--      -----
--      3858
```

En savoir plus

- [En savoir plus sur les fonctions dans Oracle SQL](#)
- [Voir tous les formats de TO_CHAR et TO_DATE](#)
- [Voir tous les formats de ROUND et TRUNC](#)

Les groupements, requetes imbriquées et opérateurs ensemblistes

Toutes ces choses ont l'avantage de permettre de faire des requêtes plus compliquées en assemblant des requêtes plus simple ou en utilisant des fonctions pour nous simplifier la vie.

☐☐ Les groupements

Les groupements permettent de grouper plusieurs résultats en même temps. Par exemple pour faire la moyenne d'une/plusieurs colonne(s), les additionner, etc.

```
-- Ici on a une fonction sum() qui permet d'additionner tous les résultats ensemble
SELECT c.description, sum(b.surface_habitable + b.surface_jardin) || 'm2' AS surface
FROM Categorie_bien c
JOIN Bien b ON b.id_catbien = c.id_catbien

-- On peut grouper les choses par description
-- (on y met juste toutes les colonnes qui sont dans le select mais pas dans la fonction de groupement)
GROUP BY c.description

-- HAVING fait la même chose qu'un WHERE mais sur le résultat d'un groupement
HAVING sum(b.surface_habitable + b.surface_jardin) BETWEEN 2000 AND 3000
```

Les fonctions de groupements que l'on peut utiliser sont :

- `AVG()` pour faire une moyenne de plusieurs résultats
- `SUM()` pour additionner plusieurs résultats
- `MAX()` pour avoir le plus grand résultat
- `MIN()` pour avoir le plus petit résultat
- `COUNT()` pour compter le nombre de résultats
 - `COUNT(*)` pour compter le nombre de tuples
 - `COUNT(attr)` pour compter le nombre de valeurs dans une colonne
 - `COUNT(DISTINCT attr)` pour compter le nombre de valeurs distinctes d'une colonne

Les requêtes imbriquées

Les requêtes imbriquées permettent de diviser des problèmes complexes en requêtes plus simples que l'on assemble entre elles.

```
-- trouvez le nom du ou des candidats qui ont fait l'offre la plus haute
SELECT c.nom
FROM candidat c
JOIN offre o ON o.id_candidat = c.id_candidat
WHERE o.montant = (SELECT max(montant) FROM offre)
```

Dans cette requête on divise le problème en deux parties :

1. Trouver le montant le plus haut (c'est la requête imbriquée)
2. Trouver le(s) candidat·e·s qui ont une offre qui correspond à ce montant

Les opérateurs ensemblistes

```
-- écrivez une requête qui affichera en 3 colonnes les noms, prénoms et rôle (Candidat, Client, Vendeur) des
candidats, des clients et des vendeurs
select nom, prenom, 'Vendeur' as role
from vendeur
union
select nom, prenom, 'Candidat' as role
from candidat
union
select nom, prenom, 'Client' as role
from client
```

Ici étant donné que l'on a les mêmes noms de colonnes pour chaque requête on peut les combiner ensemble (comme vu avec les ensembles en math ou avec les jointures)

Il existe 3 types d'opérateurs ensemblistes en Oracle SQL :

- Le **UNION** qui permet de combiner les deux (additionner en somme)
- Le **MINUS** qui va prendre tout ce qui est dans le premier mais pas après (différence en math)
- Le **INTERSECT** qui va prendre seulement ce qui est commun aux deux requêtes

Voici d'autres exemples avec d'autres opérateurs :

-- Sélectionner tous les biens qui n'ont pas fait l'objet d'une offre

```
SELECT id_bien
```

```
FROM bien
```

MINUS

```
SELECT id_bien
```

```
FROM offre
```

Ou encore

-- affichez les identifiants des biens qui ont une offre d'un montant supérieur à 200.000€ dans la province de Liège

-- On prends tous les biens qui sont dans la province de liège

```
SELECT id_bien
```

```
FROM bien
```

```
WHERE code_postal BETWEEN 4000 AND 4999
```

INTERSECT

-- Et on prends l'intersection avec toutes les offres qui ont un montant supérieur à 200000

```
SELECT id_bien
```

```
FROM offre
```

```
WHERE montant > 200000
```

Les vues et contraintes

Voici un exemple de vue qui provient d'un exercice de labo:

```
-- La vue s'appelle "chateau" et va lister tous les biens de type chateau
CREATE VIEW chateau AS

-- Ici on peut mettre la requête qui permet de lister tous les biens de type chateau
SELECT b.*
FROM Bien b
JOIN CATEGORIE_BIEN cb ON cb.ID_CATBIEN = b.ID_CATBIEN
WHERE cb.DESCRPTION = 'Château'

-- Cette ligne indique que si on veut ajouter un élément dans la vue, elle doit respecter la condition de la requête
WITH CHECK OPTION;

-- On aurait aussi pu utiliser "WITH READ ONLY" si on veut empêcher d'y insérer des données
```

Ainsi ici, on a créé un genre de table spéciale appelée "château" que l'on peut ensuite accorder l'accès à un utilisateur. Ainsi, cet utilisateur ne pourra insérer que des châteaux dans la base de donnée et ne pourra voir que la liste des châteaux et rien d'autre.

Attention, pour qu'une vue soit modifiable, il faut qu'elle soit la plus minimale possible (sans jointures, ni group by, ni opérations ensemblistes)

Contraintes

On peut aussi ajouter des contraintes directement sur la table, par exemple une clé étrangère ou encore la vérification d'une condition.

```
-- Modification de la table Alerter pour y ajouter une contrainte que statut ne peut être que L ou N
ALTER TABLE Alerter

-- La condition doit toujours être en parenthèse
ADD CONSTRAINT statutRestriction CHECK (STATUT IN ('L', 'N'));
```

Triggers

Attention, ceci est uniquement valable in SQL Server

- Créer une trigger qui renvoie une erreur dans un cas précis

```
-- On donne un nom à la trigger et on lui dit sur quelle table il opère
-- Ici le trigger opère sur la table "offre"
CREATE TRIGGER date_offre_posterieure_date_vente ON offre

-- On lui dit l'action pour laquelle il doit s'exécuter (ici après un insert)
-- On pourrait aussi avoir "after delete" ou "after update"
AFTER INSERT

AS BEGIN
    [- On déclare les variables avec leur types
    [- Ces déclarations doivent toujours être au début du trigger
    [- declare @dateOffre SMALLDATETIME
    [- declare @idBien INTEGER
    [- declare @dateMiseVente SMALLDATETIME
    [-
    [- "inserted" est une table temporaire qui renvoie le tuple inséré
    [- On pourrait aussi avoir "deleted" à la place de inserted, et dans le cas d'un UPDATE
    [- On a les deux (deleted et inserted)
    [-
    [- La syntaxe de @variable = attribut va attribuer la valeur de l'attribut du tuple
    [- dans la variable définie plus tot
    [- select @idBien = id_bien, @dateOffre = date_offre from inserted
    [-
    [- Ensuite on peut simplement utiliser la variable n'importe où dans une query
    [- select @dateMiseVente = date_mise_en_vente from bien where id_bien = @idBien
    [-
    [- on crée une condition (dans lequel il faut annuler l'insert)
    [- if @dateOffre < @dateMiseVente
    [- begin
    [- on génère une erreur
```

```

raiserror('La date de l'offre doit être postérieur à la date de la mise en vente.', 7, 1)
-- On annule l'insert
rollback transaction
-- On met fin à la trigger
return
end

-- si la condition ne correspond pas, alors l'insert est bien inséré
END

```

- On peut aussi faire un curseur pour itérer sur plusieurs lignes d'une requête SQL

```

-- Ici on déclare une variable curseur pour une certaine query (que l'on va itérer)
DECLARE crsrClientA_F CURSOR
FOR SELECT nom, adresse
FROM Client
WHERE UPPER(SUBSTRING(nom,1,1)) IN ('A','B','C','D','E','F')

-- On ouvre le curseur et on récupère le premier élément dans des variables
OPEN crsrClientA_F
FETCH crsrClientA_F INTO @nom, @adresse

-- Tant que le dernier fetch a trouvé quelque chose, on va print le nom et l'adresse
WHILE @@FETCH_STATUS = 0
BEGIN
PRINT @nom + ' habite à ' + @adresse

-- A la fin de la boucle, on refait un fetch pour récupérer la valeur suivante
-- La boucle s'arrêtera quand ce fetch ratera
FETCH crsrClientA_F INTO @nom, @adresse
END

-- Une fois terminé on peut fermer le curseur et le désallouer
CLOSE crsrClientA_F
DEALLOCATE crsrClientA_F

```