Structured Query Language

SQL : l'essentiel (ou presque)

Alexis NEDELEC

Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

Introduction

Historique

Langages pour décrire et manipuler des données relationnelles

- QUEL (INGRES,université de Berkeley)
- QBE, SQL (IBM)

Accès aux SGBD Relationnels

- Langage de Description de Données (LDD)
- Langage de Manipulation de Données (LMD)

Structured Query Language

SQL : Langage “universel” pour

- Concevoir, créer des ensembles de données (LDD)
- Manipuler des informations inter-dépendentes (LMD)

Structuration de données (python)

Structures linéaires, séquences (listes,dictionnaires ...)

```
points=[
        (0.0,0.0),
        (0.0,100.0),
        (100.0,100.0),
        (100.0,0.0)
       ]
```

Recherche dans une liste

```
query=[]
for point in points :
    if point[0]==100:
        query.append(point)
print(query)
```

Structuration de données (SQL)

Structuration tabulaire (lignes,colonnes)

```
CREATE TABLE points(x float, y float);
INSERT INTO points VALUES(0.0,0.0);
INSERT INTO points VALUES(0.0,100.0);
INSERT INTO points VALUES(100.0,100.0);
INSERT INTO points VALUES(100.0,0.0);
```

Recherche dans une table

```
SELECT *
FROM points
WHERE x=100;
```

Persistance de données (python)

Python : gestion de fichiers

- fonction `open()` : lecture ligne à ligne
- module `cPickle` : fonctions `dump(),load()`
- module `json` : format JavaScript Object Notation

Exemple de sauvegarde au format json

```
import json
points_to_save=[(0.0,0.0),(0.0,100.0), ...]
with open("points.json","w") as f:    # "w" : ecriture
    json.dump(points_to_save,f)
points_to_load=[]
with open("points.json","r") as f:    # "r" : lecture
    points_to_load=json.load(f)
```

Persistance de données (Python/SQL)

Python : connexion sur une base de données

```
import sqlite3
points_to_save=[(0.0,0.0),(0.0,100.0), ...]
connect = sqlite3.connect("points.db")
cursor = connect.cursor()
cursor.execute("DROP TABLE IF EXISTS points")
cursor.execute("CREATE TABLE points(x float, y float)")
cursor.executemany("INSERT INTO points VALUES (?,?)",
                   points_to_save)
connect.commit()
```

Persistance de données (Python/SQL)

Recherche d'informations

```
points_to_load=[]
for point in cursor.execute("SELECT * FROM points")  :
    points_to_load.append(point)
```

Insertion d'informations

```
cursor.execute("INSERT INTO points VALUES(50,-50)")
cursor.execute("INSERT INTO points VALUES(100,0)")
```

Mise à jour d'informations

```
cursor.execute("UPDATE points SET y=-100 WHERE y=-50")
```

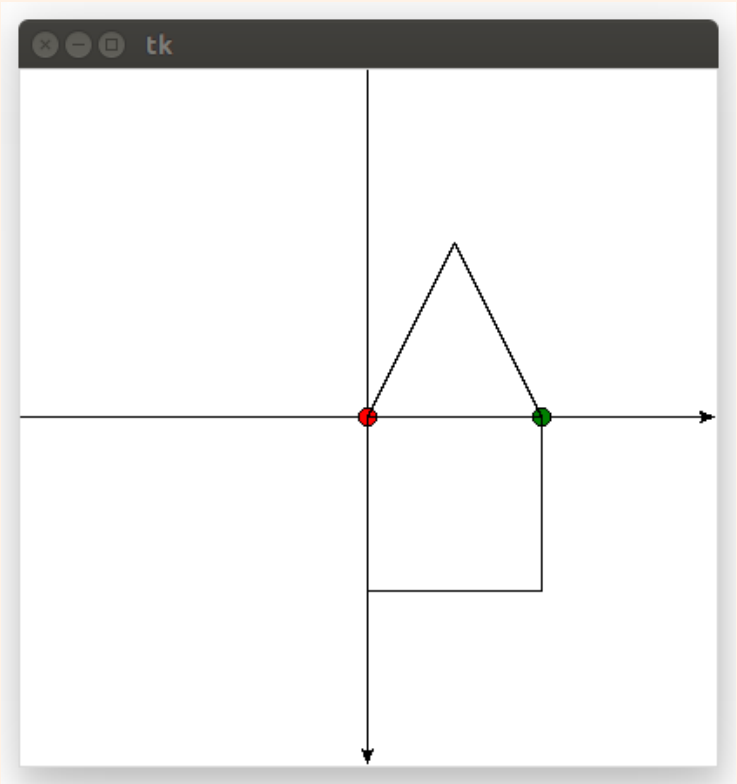
Persistance de données (Python/SQL)

Application graphique (TkInter) : affichage des points

```
from tkinter import Tk,Canvas
root=Tk()
w,h=400,400
canvas = Canvas(root,
                width=w,height=h,background="white")
# affichage de repere au centre du Canvas (h/2,w/2)
canvas.create_line(0, h/2.0, w, h/2.0, arrow="last")
canvas.create_line(w/2.0, 0, w/2.0,h,arrow="last")
# TODO : recuperer les points dans ce repere
# TODO : visualiser les extremites de la polyligne
# TODO : afficher les points de la polyligne
canvas.pack()
root.mainloop()
```

Persistance de données (Python/SQL)

Application graphique (`TkInter`) : affichage des points



Code complet de l'application en annexe (p. **??**)

Langage SQL

Normalisation ANSI/ISO (www.iso.org)

- SQL 1 (1986) : standard de description/manipulation
- SQL 2 (1992) : supporte totalement le modèle relationnel
- SQL 3 (1999) : doit supporter les modèles objets,
- SQL 4 (200 ?) : modèle déductif...

Objectifs principaux

- Création du schéma de la base (tables et associations)
- Recherche d'informations (jointures entre tables)
- Modification d'informations (manipulation des n-uplets)

Langage SQL

Gestion de données : langage déclaratif SQL

- DDL : Data Definition Language
- DML : Data Manipulation Language
- DCL : Data Control Language
- TCL : Transaction Control Language

Traitement de données : langage procédural PL/SQL

- PSM : Persistent Stored Module (triggers ...)
- CLI : Call Level Interface (ODBC ...)
- Embedded SQL : ordre SQL dans un langage hôte

SQL Déclaratif

Préciser quoi sans savoir comment y accéder

Opérations de bases

- **Recherche** (SELECT en SQL / RETRIEVE en QUEL)
- **Insertion** (INSERT en SQL / APPEND en QUEL)
- **Suppression** (DELETE en SQL / SUPPRESS en QUEL)
- **Modification** (UPDATE en SQL / REPLACE en QUEL)

Recherche d'informations en SQL

`SELECT sur attributs de n-uplets`	(projection, Π)
`FROM relation(s)`	(produit cartésien, $\times$)
`WHERE qualification`	(restriction, σ)

Définition de données (DDL)

Commandes de création

- `CREATE SCHEMA` : Espace de nommage, création de droits ...
- `CREATE TABLE` : création d'une entité structurée
- `CREATE VIEW` : "encapsulation" de requêtes dans une vue
- `ALTER, DROP TABLE` : modification, destruction de tables
- `CREATE INDEX` : efficacité de recherche sur des colonnes
- `CREATE TYPE, DOMAIN` : création de types de données
- `CREATE FUNCTION` : codé en SQL procédural
- `CREATE PROCEDURE` : procédures stockées
- `CREATE TRIGGER` : déclencheurs sur modification de tables

Contraintes sur les données (DDL)

Contraintes de colonnes

- `DEFAULT` : valeur par défaut dans une colonne
- `NOT NULL` : contrainte de colonne non vide
- `UNIQUE` : pas de doublons dans une colonne
- `PRIMARY KEY` : clé primaire (sur 1 ou plusieurs colonnes)
- `FOREIGN KEY` : référence sur une colonne de table “mère”
- `CHECK` : contrainte de validité sur une colonne
  - `BETWEEN` : plage de valeurs
  - `LIKE,SIMILAR,_,%` : comparaison partielle
  - `IN` : liste de valeurs possibles
  - `CASE` : branchement sur des valeurs possibles
  - `OVERLAPS` : recouvrement de périodes

Contraintes sur les données (DDL)

Valeurs par défaut

```
DROP TABLE IF EXISTS personnes;

CREATE TABLE personnes (id integer,nom text);
INSERT INTO personnes(id,nom) VALUES(1,'Dupond');
SELECT * FROM personnes;

ALTER TABLE personnes
  ADD COLUMN age SMALLINT DEFAULT 18;
INSERT INTO personnes(id,nom) VALUES(1,'Dupont');
SELECT * FROM personnes;
```

Contraintes sur les données (DDL)

Contraintes sur l'insertion de valeurs

```
INSERT INTO personnes(nom) VALUES('Dupond');
ALTER TABLE personnes ALTER COLUMN id SET NOT NULL;
DELETE FROM personnes WHERE id IS NULL;
ALTER TABLE personnes ALTER COLUMN id SET NOT NULL;
```

Contraintes d'unicité de valeurs

```
ALTER TABLE personnes
  ADD CONSTRAINT personnes_id_unique UNIQUE(id);
DELETE FROM personnes WHERE nom='Dupont';
ALTER TABLE personnes
  ADD CONSTRAINT personnes_id_unique UNIQUE(id);
```

Contraintes sur les données (DDL)

Exercice

- créer une table `t1(c1:integer,c2:text)`
- insérer des enregistrements
  `(NULL,'Toto'),(1,'Toto'),(1,'Titi')`
- définir une contrainte de colonne non-nulle sur `c1`
- modifier l'enregistrement `(1,'Titi')` en `(2,'Titi')`
- définir une contrainte d'unicité `t1_c1_unique` sur `c1`
- renommer la table `t1` en `personne`
- renommer la colonne `c1` en `id` et `c2` en `nom`
- renommer contrainte d'unicité en `personnes_id_unique`
- remplacer `'Toto','Titi'` par `'Dupond','Dupont'`
- créer une séquence `personnes_id_seq` sur la colonne `id`

Manipulation de données (DML)

Création, lecture, mise à jour, destruction de données

- `CREATE TABLE` : création d'un ensemble (table)
- `SELECT ... FROM ... WHERE ...` : recherche d'élements
- `INSERT INTO ... VALUES (...)` : insertion d'élements
- `UPDATE ... SET ... WHERE ...` : modification d'élements
- `DELETE FROM ... WHERE ...` : destruction d'élements

Manipulation de Données (DML)

Exemples : gestion de personnes

- `CREATE TABLE personnes (id integer,nom text);`
- `SELECT * FROM personnes WHERE nom='Dupond';`
- `INSERT INTO personnes(id,nom) VALUES(1,'Dupond');`
- `UPDATE personnes SET nom='Dupont' WHERE id=1;`
- `DELETE FROM personnes WHERE id=2;`

Consultation de personnes

```
SELECT * FROM personnes WHERE id=1;
 id |  nom
----+--------
  1 | Dupont
(1 ligne)
```

Manipulation de Données (DML)

Recherche de données

```
SELECT [DISTINCT] <liste de colonnes>
FROM   <liste de tables>
[WHERE  <conditions de recherche>]
[GROUP BY  <partitionnement horizontal>]
[HAVING <conditions de recherche sur les groupes>]
```

Exercice : recherche de `personnes`

- rechercher toutes les personnes, rechercher les `'Dupont'`
- rajouter une colonne `age`
- mettre à jour l'`age` de `'Dupont'` à `20`
- insérer : ('Dupond',21), ('Durand',20), ('Durant',30)
- donner, par age, le nombre de personnes de moins de 30 ans

Manipulation de Données (DML)

Ordre d'éxécution de requête

1. `FROM` : concaténation de chaque ligne de chaque table.
2. `WHERE` : élimination des lignes ne vérifiant pas les conditions (`FALSE,UNKNOWN`) sur la table de travail.
3. `GROUP BY` : répartition des lignes résultantes dans des groupes où les valeurs dans une même colonne sont identiques
4. `HAVING` : restriction des lignes vérifiant les conditions (`TRUE`) sur les regroupements.
5. `SELECT` : élimination des colonnes non mentionnées.
6. `DISTINCT` : élimination des lignes dupliquées.
   NB : Si `GROUP BY` le `DISTINCT` est inutile.

Manipulation de Données (DML)

Résultat de la recherche

- `ALL` : on garde tout
- `DISTINCT` : on élimine les doublons
- `ORDER BY` : tri de résultat
  - noms de colonnes : par ordre de "niveau de détails"
  - `ASC, DESC` : tri ascendant ou descendant par colonne
  - `COLLATE` : sensible à la casse, aux accents ... par colonne

Formulation de requête

Sur une table

```
SELECT * FROM personnes ORDER BY nom DESC;
 id |  nom
----+--------
  1 | Zupond
  1 | Dupond
(2 lignes)
```

Formulation de requête

Opérations ensemblistes

Entre résultats de deux requêtes

- `UNION` : concatène les 2 requêtes
- `INTERSECT` : les données communes aux 2 requêtes
- `EXCEPT` : les données qui ne sont pas dans l'autre requête

Exemples : gestion de personnes

```
(SELECT  nom FROM personnes)
INTERSECT
(SELECT  nom FROM employes);
```

Cette requête retournera :

- le nom des personnes **qui sont** des employés

Formulation de requête

Recherches imbriquées

- `IN (=ANY)` : appartenance ($\in$) à un ensemble

Exemples : gestion de personnes

```
SELECT nom
FROM personnes
WHERE nom IN (
              SELECT nom
              FROM employes
             );
```

Cette requête retournera :

- le nom des personnes **qui sont** des employés

Formulation de requête

Recherches imbriquées

- `EXISTS,` : existence ($\exists$) d'un élément dans un ensemble

Exemples : gestion de personnes

```
SELECT p.nom
FROM personnes p
WHERE EXISTS (
                SELECT *
                FROM employes e
                WHERE p.nom=e.nom
              );
```

Cette requête retournera :

- le nom des personnes **qui sont** des employés

Formulation de requête

Recherches imbriquées

- `NOT IN (<>ALL)` : non-appartenance ($\notin$) à un ensemble

Exemples : gestion de personnes

```
SELECT nom
FROM personnes
WHERE nom NOT IN (
                    SELECT nom
                    FROM employes
                  );
```

Cette requête retournera :

- le nom des personnes **qui ne sont pas** des employés

Cette requête pourra aussi être formulée par l'opération ensembliste de différence (`EXCEPT`)

Formulation de requête

Recherches imbriquées

- `NOT EXISTS` : non-existence ($\nexists$) d'un élément dans un ensemble

Exemples : gestion de personnes

```
SELECT p.nom
FROM personnes p
WHERE NOT EXISTS (
                  SELECT *
                  FROM employes e
                  WHERE p.nom=e.nom
                 );
```

Cette requête retournera :

- le nom des personnes **qui ne sont pas** des employés

Formulation de requête

Requêtes imbriquées : retour de valeur simple

```
SELECT *
FROM T1
WHERE T1.a = (SELECT COUNT(*) FROM T2 )
```

Pour chaque élément de $T1$ vérifier que la valeur de la colonne $T1.a$ est égale au nombre d'éléments de la table $T2$

Requêtes imbriquées : retour d'un ensemble de valeurs

```
SELECT *
FROM T1
WHERE T1.a IN (SELECT T2.b FROM T2 )
```

Pour chaque élément de $T1$ vérifier que la valeur de la colonne $T1.a$ est dans l'ensemble des valeurs de la colonne b de la table $T2$

Formulation de requête

Jointures entre tables

- `CROSS JOIN, NATURAL JOIN`
- `INNER JOIN,OUTER JOIN (LEFT,RIGHT,FULL)`

Exemples : gestion de personnes

```
SELECT p.nom
FROM personnes p CROSS JOIN employes e
WHERE p.nom=e.nom;

SELECT p.nom
FROM personnes p
INNER JOIN employes e ON (p.nom=e.nom);
```

Cette requête retournera :

- le nom des personnes **qui sont** des employés

Formulation de requête

Fonctions de calcul

- `COUNT,SUM` : nombre, somme d'occurences
- `MIN,MAX,AVG` : valeur min,max et moyenne
- `STDEV_POP,STDEV_SAMP` : écart-type de population,échantillon
- `VAR_POP,VAR_SAMP` : variance de population,échantillon

Exemples : gestion de personnes

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

Formulation de requête

Exemples : gestion de personnes

```
SELECT age, COUNT(*)
FROM personnes
GROUP BY age;
```

Cette requête retournera :

- le nombre de personnes selon leur âge

Exemples : gestion de personnes

```
SELECT age, COUNT(*)
FROM personnes
GROUP BY age
HAVING count(id) > 1;
```

Cette requête retournera :

- le nombre de personnes selon leur âge s'ils sont au moins deux

Formulation de requête

Exemples : gestion de personnes

```
SELECT age, COUNT(*)
FROM personnes
WHERE nom='Dupont'
GROUP BY age
HAVING Count(id) > 1;
```

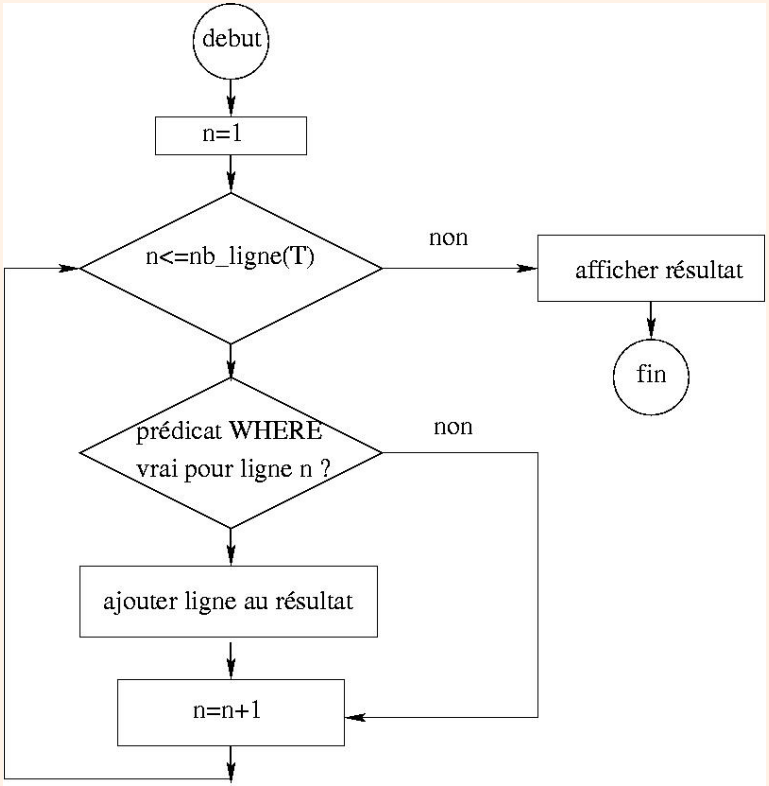
Cette requête retournera :

- le nombre de 'Dupont' selon leur âge s'ils sont au moins deux

Exécution de requête

Opérations sur une table

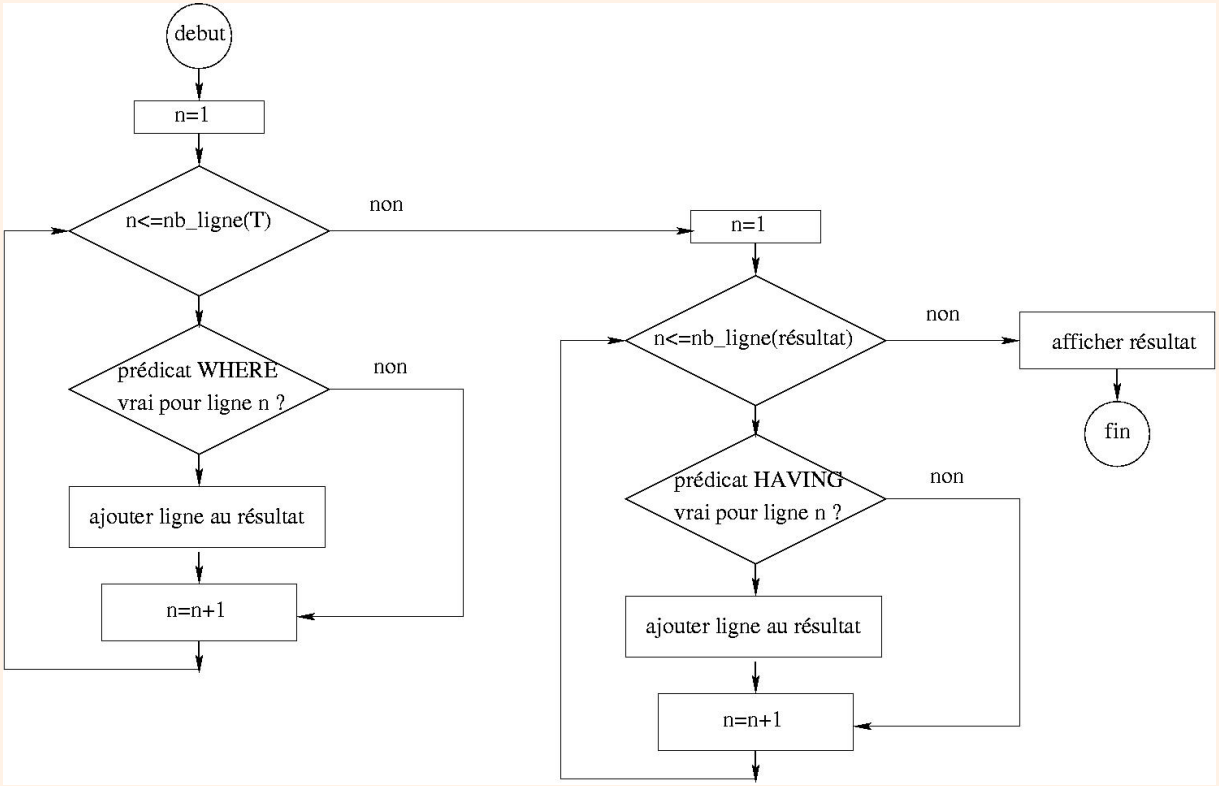
- `SELECT` : projection sur colonnes (Π)
- `FROM` : produit cartésien entre tables ($\times$)
- `WHERE` : restriction sur les enregistrements (σ)



Exécution de requête

Opérations sur une table avec groupement

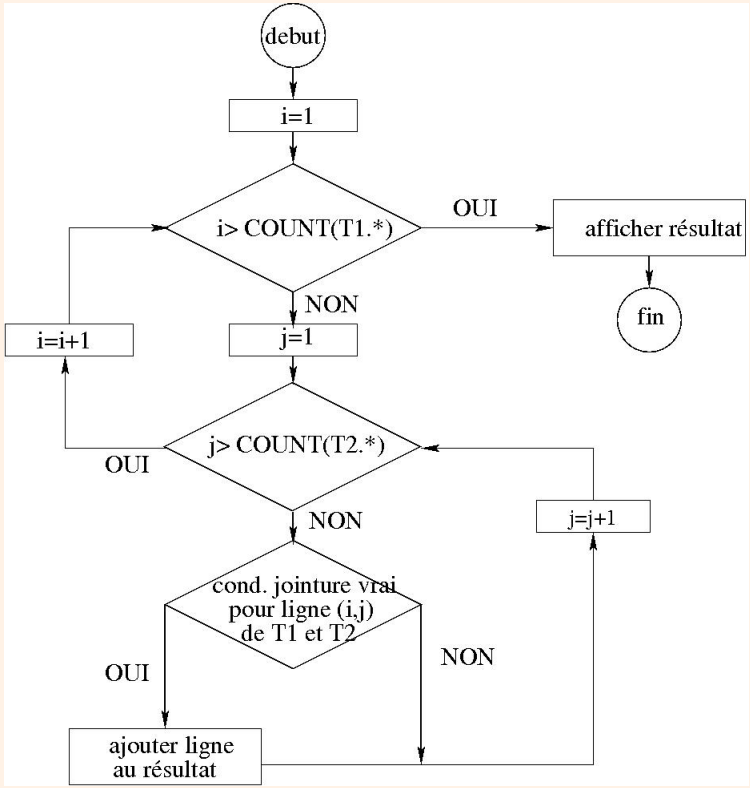
- `GROUP BY` : partitionnement du résultat
- `HAVING` : restriction sur les partitionnements du résultat



Exécution de requête

Opérations sur deux tables

```
SELECT *                    (resultat)
FROM T1 CROSS JOIN T2       (produit cartesien,jointure)
WHERE T1.a=T2.b             (condition de jointure)
```



SQL et notation mathématique

Correspondance de symboles

- `IN` : appartenance ($\in$) à un ensemble
- `NOT` : opérateur unaire de négation ($\neg$)
- `EXISTS` : quantificateur existentiel ($\exists$)
- `UNION,INTERSECT,EXCEPT` : opérateurs ensemblistes ($\cap, \cup, -$)
- `AND,OR` : connecteurs logiques ($\wedge, \vee$)

`https://fr.wikipedia.org/wiki/Connecteur_logique`

Types de données (DDL)

Chaînes de caractères

- `CHAR` : taille fixe codé sur 1 octet (ASCII, EBCDsIC)
- `VARCHAR` : taille variable codé sur 1 octet
- `NCHAR` : taille fixe codé sur 2 octets (Unicode)
- `NVARCHAR` : taille variable codé sur 2 octets (Unicode)
- `CLOB,NCLOB` : grande taille (Character Large OBject)

Types de données (DDL)

Datation

- `DATE` : date (jour,mois,année)
- `TIME` : heure
- `TIMESTAMP` : date et heure
- `TIME (TIMESTAMP) WITH TIME ZONE` : avec fuseau horaire
- `INTERVAL` : intervalle de temps, durée

Date de naissance d'une personne

```
ALTER TABLE personnes DROP COLUMN IF EXISTS naissance;
SELECT * FROM personnes;
ALTER TABLE personnes
  ADD COLUMN naissance DATE DEFAULT
    CURRENT_DATE - interval '18 years';
```

Types de données (DDL)

Types entiers

- `INTEGER` : entier sur 32 bits (en principe)
- `SMALLINT` : entier sur 16 bits (< `INTEGER`)
- `BIGINT` : nouveauté SQL :2003 pour 64 bits (> `INTEGER`)

Types réels : approximation

- `FLOAT` : on peut fixer la précision
- `REAL` : sans fixer de précision
- `DOUBLE PRECISION` : plage de valeurs (> `REAL`)

Types de données (DDL)

Types décimaux : valeur exacte

- `NUMERIC[(n[,p])]` : sur n chiffres, p chiffres après la virgule
- `DECIMAL[(n[,p])` : id `NUMERIC`, interne au SGBDR

Types binaires

- `BOOLEAN` : booléen
- `BIT[(n)]` : chaîne de bits de taille fixe (n bits)
- `VARBIT[(n)]` : chaîne de bits de taille variable (n bits)
- `BLOB` : Binary Large OBject (en K,M,G octets)

Types de données (DDL)

Opérateurs de comparaison

Op.	Numérique	Caractère	Date
<	inférieur	classé avant	plus tôt que
=	égal	équivalent à	en m̂ temps que
>	supérieur	classé après	plus tard que
<=	inf. ou égal	classé avant ou équ.	pas plus tard que
<>	non égal	différent de	pas en m̂ temps que
>=	sup. ou égal	classé après ou équ.	pas plus tôt que

Conclusion

SQL : Intérêts

- Langage de Description de Données
- Langage de Manipulation de Données
- Langage standard pour les Bases de Données Relationnelles
- normalisation successives (`SQL1, SQL2, SQL3 ...`)

SQL : Inconvénients

- pauvreté de la modélisation :
  - un seul type de lien (`FOREIGN KEY`)
  - association par jointure (`INNER,OUTER,NATURAL`)
- pauvreté de la représentation : tabulaire, types atomiques
- déclaratif vs programmation (impedence mismatch)
- syntaxe : optimisation de requêtes à l'écriture

Bibliographie

Adresses “au Net”

- PostgreSQL : `www.postgresql.org`
- SQLite : `https://www.sqlite.org`
- Cours et tutoriaux sur SQL : `https://sql.sh`
- Tutoriaux W3C sur SQL : `http://www.w3schools.com/sql`
- Site de Georges Gardarin : `georges.gardarin.free.fr`
- Cours de Maude Manouvrier : `www.lamsade.dauphine.fr/~manouvri`

Sans oublier le CRD ENIB : `https://wiki.enib.fr/crd`

Annexe : Application graphique

Canvas : widget d'affichage TkInter

```
from tkinter import Tk,Canvas
root=Tk()
w,h=400,400
canvas = Canvas(root,
                width=w,height=h,background="white")
# affichage de repere au centre du Canvas (h/2,w/2)
canvas.create_line(0, h/2.0, w, h/2.0, arrow="last")
canvas.create_line(w/2.0, 0, w/2.0,h,arrow="last")
# TODO : recuperer les points dans ce repere
# TODO : visualiser les extremites de la polyligne
# TODO : afficher les points de la polyligne
canvas.pack()
root.mainloop()
```

Annexe : Application graphique

Récupération des données persistantes

```
conn = sqlite3.connect("points.db")
cursor = conn.cursor()
points_to_draw=[]
for point in  cursor.execute("SELECT * FROM points") :
    points_to_draw.append(
                            (
                             w/2.0 + point[0],
                             h/2.0 + point[1]
                            )
                           )
```

Annexe : Application graphique

Canvas : widget d'affichage TkInter

```
# TODO : visualiser les extremites de la polyligne
    start_x=points_to_draw[0][0]
    start_y=points_to_draw[0][1]
    # starting point of polyline : red color
    start_id=canvas.create_oval(
                                    start_x-5,start_y-5,
                                    start_x+5,start_y+5,
                                    fill="red"
                                   )
```

Annexe : Application graphique

Canvas : widget d'affichage TkInter

```
    end_x=points_to_draw[len(points_to_draw)-1][0]
    end_y=points_to_draw[len(points_to_draw)-1][1]
    # ending point of polyline  : green color
    end_id=canvas.create_oval(
                                end_x-5,end_y-5,
                                end_x+5,end_y+5,
                                fill="green"
                               )
#  TODO : afficher les points de la polyligne
    polyline_id=canvas.create_line(points_to_draw)
```