Bases de Données Avancées - Partie 1 : SQL (Structured Query Language)

Thomas Gerald

September 20, 2024

Laboratoire Interdisciplinaire des Sciences du Numérique – LISN, CNRS
thomas.gerald@lisn.upsaclay.fr

Partie 1 : SQL (Structured Query Language)

- Le SQL c'est quoi ?
- Définir une relation avec SQL
- Les requêtes SQL

SQL, c'est quoi ?

Un langage d'interaction avec les bases de données relationnelles
→ Originellement développés par IBM (projets SEQUEL-XRM et System-R, 1974-1977)
→ Le langage le plus utilisé pour l'interaction avec des bases de données relationnelles
→ Interaction avec des données tabulaires (données sous forme de table)

Structured Query Language (SQL) : À quel Niveau

On ne s'intéresse pas au stockage des données mais plutôt à un niveau conceptuel des données (**Niveau Logique**)!

On ne s'intéresse pas au stockage des données mais plutôt à un niveau conceptuel des données (Niveau Logique)!

Au niveau Logique

→ On ne s'intéresse pas à la structure de stockage !!!

Différentes parties du langage

- La définition des données (Data definition Langage - **DDL**) Un sous ensemble pour la création, la suppression et la modification des définitions des tables et des vues

Différentes parties du langage

- La définition des données (Data definition Langage - **DDL**) Un sous ensemble pour la création, la suppression et la modification des définitions des tables et des vues
- Manipulation des données (Data Manipulation Language - **DML**) Un sous ensemble du langage permettant de manipuler les données en autorisant l'insertion (`INSERT`), la suppression (`DELETE`), la modification (`UPDATE`)

Différentes parties du langage

- La définition des données (Data definition Langage - **DDL**) Un sous ensemble pour la création, la suppression et la modification des définitions des tables et des vues
- Manipulation des données (Data Manipulation Language - **DML**) Un sous ensemble du langage permettant de manipuler les données en autorisant l'insertion (`INSERT`), la suppression (`DELETE`), la modification (`UPDATE`)
- La sécurité des données Contrôle de l'accès des données (utilisateurs, etc...)

Différentes parties du langage

- La définition des données (Data definition Langage - **DDL**) Un sous ensemble pour la création, la suppression et la modification des définitions des tables et des vues
- Manipulation des données (Data Manipulation Language - **DML**) Un sous ensemble du langage permettant de manipuler les données en autorisant l'insertion (`INSERT`), la suppression (`DELETE`), la modification (`UPDATE`)
- La sécurité des données Contrôle de l'accès des données (utilisateurs, etc...)
- Gestion des transactions Des commandes explicites pour contrôler les transactions

Différentes parties du langage

- La définition des données (Data definition Langage - **DDL**) Un sous ensemble pour la création, la suppression et la modification des définitions des tables et des vues
- Manipulation des données (Data Manipulation Language - **DML**) Un sous ensemble du langage permettant de manipuler les données en autorisant l'insertion (`INSERT`), la suppression (`DELETE`), la modification (`UPDATE`)
- La sécurité des données Contrôle de l'accès des données (utilisateurs, etc...)
- Gestion des transactions Des commandes explicites pour contrôler les transactions
- ...

Tables et SQL

Une table SQL est un objet (une abstraction) qui contient les enregistrements d'une relation.

etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

- **Le nom de la relation** : etu (pour étudiant)
- **Les champs/colonnes** : nom, prenom, mail
- **Un enregistrement** : Une ligne de la table

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **`str`**, *prenom*: **`str`**, *mail*: **`str`**)

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *nom* et *prenom* de type **(str, str)** (tuple)

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *nom* et *prenom* de type **(str, str)** (tuple)

Utilisation de chaîne de caractères pour les clefs primaires?
→ Plus de stockage pour les clefs étrangères
→ ...

Etu		
nom	prenom	mail
Zeblouse	Agathe	az@*****
Huai	Odile	oh@*****
Peuplu	Jean	jp@*****
Hochon	Paul	hp@*****
⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu(*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *nom* et *prenom* de type **(str, str)** (tuple)

Utilisation de chaîne de caractères pour les clefs primaires?
→ Plus de stockage pour les clefs étrangères
→ ...

On préfère dédier un champs numérique pour les clefs primaires (mais ce n'est pas une obligation)

Etu			
eid	nom	prenom	mail
1	Zeblouse	Agathe	az@*****
2	Huai	Odile	oh@*****
3	Peuplu	Jean	jp@*****
4	Hochon	Paul	hp@*****
⋮	⋮	⋮	⋮

Quelles sont les clefs primaires possibles ?

Etu(*eid*: **int**,*nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *mail* de type **str** (chaîne de caractères)

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *nom* et *prenom* de type **(str, str)** (tuple)

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
→ Clef primaire sur le champs *eid* de type **int**

Le Language de Définition des Données (LDD)

On utilise le **Langage de définition des données**, pour définir le schéma des relations

- **CREATE** : Création du schéma de la relation
- **ALTER** : Modification du schéma de la relation
- **DROP** : Suppression d'une relation

On souhaite créer la table `etu` avec le schéma ci-dessous :

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)

Comment faire ?

On souhaite créer la table **etu** avec le schéma ci-dessous :

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)

Comment faire ?

```
CREATE TABLE etu( -- nom de la table
   eid INTEGER, -- un entier
   nom VARCHAR(32), -- une string
   prenom VARCHAR(32),
   mail VARCHAR(128),
)
```

On souhaite créer la table **etu** avec le schéma ci-dessous :

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)

```
CREATE TABLE etu( -- nom de la table
   eid INTEGER, -- un entier
   nom VARCHAR(32), -- une string
   prenom VARCHAR(32),
   mail VARCHAR(128),
)
```

Comment faire ?

Et pour spécifier la clef primaire ?

On souhaite créer la table **etu** avec le schéma ci-dessous :

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)

Comment faire ?

```
CREATE TABLE etu( -- nom de la table
   eid INTEGER, -- un entier
   nom VARCHAR(32), -- une string
   prenom VARCHAR(32),
   mail VARCHAR(128),
)
```

Et pour spécifier la clef primaire ? → Ajouter une contrainte

On peut spécifier la clef primaire, en ajoutant une contrainte:

→ Au moment de la définition:

CONSTRAINT	object_key_contraint	PRIMARY KEY	(x)
↓	↓	↓	↓
Ajout d'une contrainte	Le nom de la contrainte	le type de contrainte	le champ

```
CREATE TABLE etu( -- nom de la table
   eid INTEGER, -- un entier
   nom VARCHAR(32), -- une chaîne de caractères
   prenom VARCHAR(32),
   mail VARCHAR(128),
   CONSTRAINT pk_etu PRIMARY KEY (nom, prenom) -- contrainte de clef primaire sur
      -- le couple (nom,prenom) nommé pk_etu
)
```

Dans la suite on va plutôt considérer le schéma suivant pour la relation **etu**

Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)

```
CREATE TABLE etu( -- nom de la table
   eid INTEGER, -- un entier
   nom VARCHAR(32), -- une chaîne de caractères
   prenom VARCHAR(32),
   mail VARCHAR(128),
   CONSTRAINT pk_etu PRIMARY KEY (eid) -- contrainte de clef primaire sur eid
)
```

On considère deux nouvelles relations définit par:

→ Cours(*cid*: **int**, *titre*: **str**)
→ Eval(*nid*: **int**, *etu_id*: **int**, *cours_id*: **integer**, *note*: **float**)

où le champ **etu_id** fait référence à la clef primaire de la table **Etu** :

- Un étudiant peut avoir plusieurs notes
- Si on ajoute un enregistrement dans **eval** alors la valeur **etu_id** doit exister dans **Etu** (eid existant)

Il s'agit du concept de clef étrangère (Notons que cours pourrait aussi être une clefs étrangère sur une table)

On peut spécifier la clef étrangère, en ajoutant une contrainte:

→ Au moment de la définition:


```
CREATE TABLE eval( -- nom de la table
   nid INTEGER, -- un entier
   etu_id INTEGER, -- une chaîne de caractères
   cours_id INTEGER,
   note FLOAT,
   CONSTRAINT pk_note PRIMARY KEY (nid), -- contrainte de clef primaire
   CONSTRAINT fk_note_etu FOREIGN KEY(etu_id) REFERENCES etu(eid)
)
```

⚠ Pour la clef étrangère, le champ référencé doit être unique

→ **eid** doit être unique dans **etu**

Structured Query Language (SQL) : autres contraintes

On peut contraindre certains champs à certaines valeurs, unicité...
Si l'on considère les relations créées précédemment

- Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **int**, *titre*: **str**)
- Eval(*nid*: **int**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Structured Query Language (SQL) : autres contraintes

On peut contraindre certains champs à certaines valeurs, unicité...
Si l'on considère les relations créées précédemment

- Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **int**, *titre*: **str**)
- Eval(*nid*: **int**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Quelles contraintes vous semblent légitimes pour la relation Eval ?

Structured Query Language (SQL) : autres contraintes

On peut contraindre certains champs à certaines valeurs, unicité...
Si l'on considère les relations créées précédemment

- Etu(*eid*: **int**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **int**, *titre*: **str**)
- Eval(*nid*: **int**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Quelles contraintes vous semblent légitimes pour la relation Eval ?

- Clef étrangère sur cours_id
- Une contrainte d'unicité sur le couple cours/étudiant (si une note par cours)
- Une contrainte sur la valeur `note` (entre 0 et 20 par exemple)

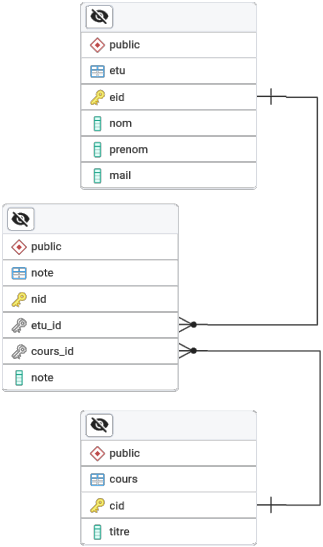
Notons que le couple cours_id, etu_id pourrait être une clef primaire

Structured Query Language (SQL) : Autres contraintes

- Clef étrangère sur cours_id
- Une contrainte d'unicité sur le couple cours/étudiant (si une note par cours)
- Une contrainte sur la valeur `note` (entre 0 et 20 par exemple)
- Pas de valeur nulles sur note (pas forcément)

```
CREATE TABLE eval( -- nom de la table
   nid INTEGER, -- un entier
   etu_id INTEGER, -- une chaîne de caractère
   cours_id INTEGER,
   note FLOAT,
   CONSTRAINT pk_note PRIMARY KEY (nid), -- contrainte de clef primaire
   CONSTRAINT fk_note_etu FOREIGN KEY(etu_id) REFERENCES etu(eid),
   CONSTRAINT fk_note_cours FOREIGN KEY(cours_id) REFERENCES cours(cid),
   CONSTRAINT u_note_etu_cours UNIQUE(etu_id, cours_id),
   CONSTRAINT c_note CHECK (note >= 0 AND note <= 20),
   CONSTRAINT nn_note CHECK (note IS NOT NULL)
)
```

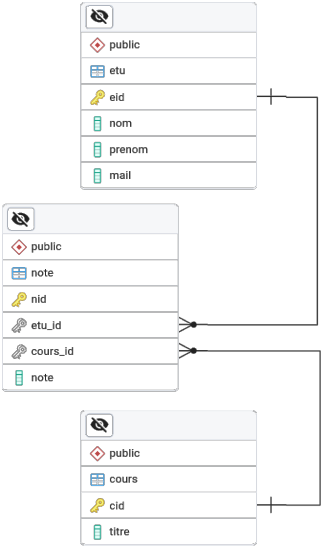
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours

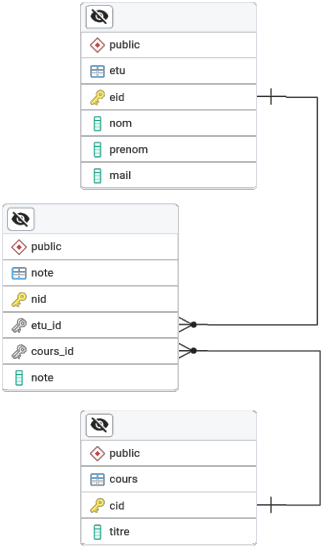
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ❌

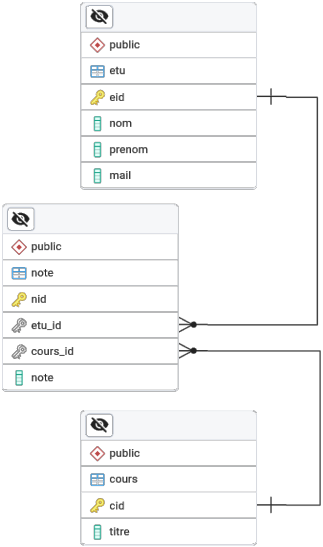
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval

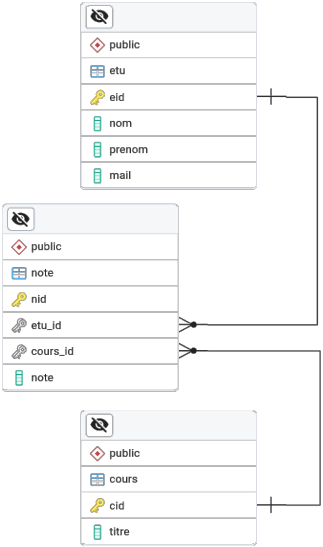
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓

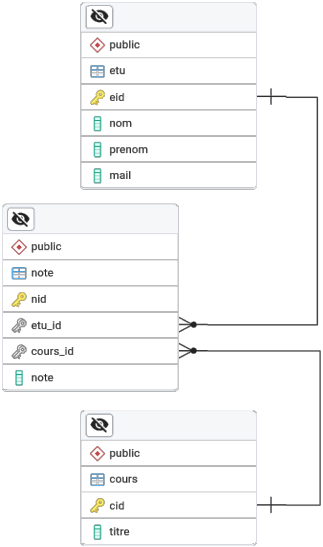
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu

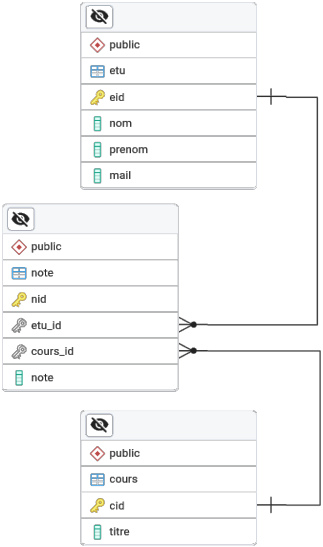
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗

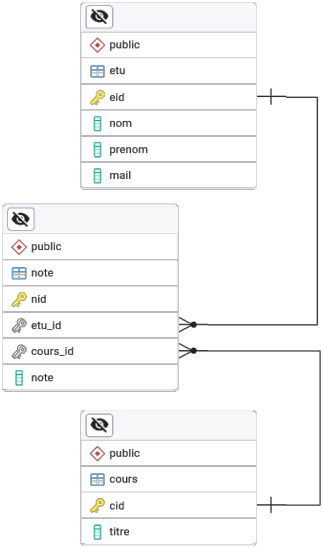
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours

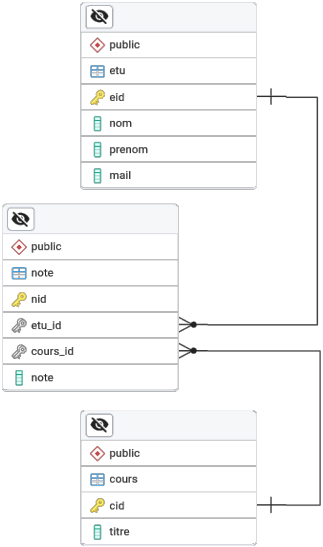
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗

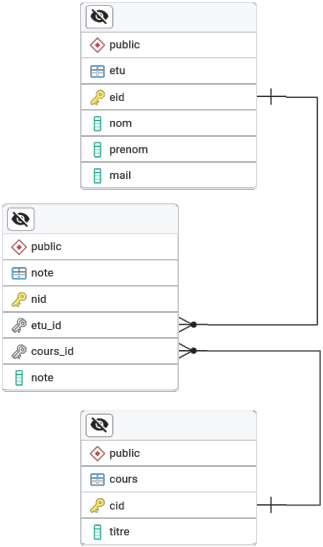
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval

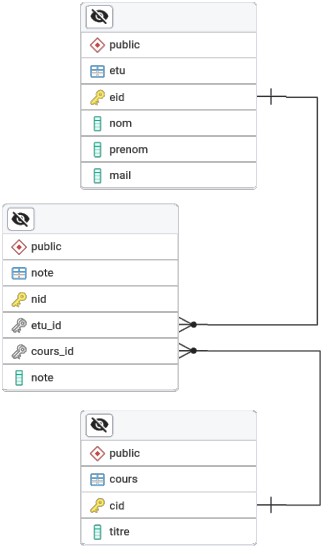
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval ✓

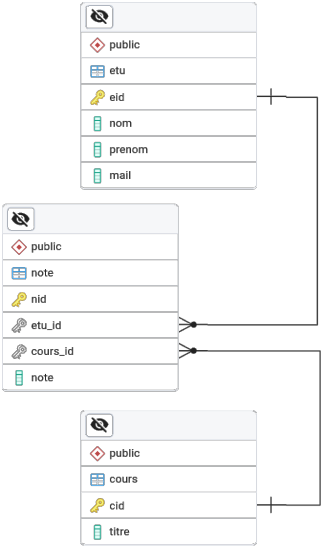
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval ✓
- Cours, Eval, Etu

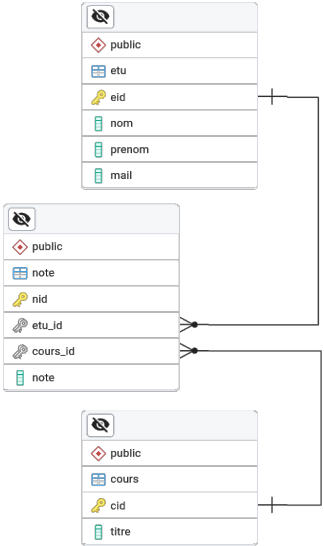
Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval ✓
- Cours, Eval, Etu ✗

Structured Query Language (SQL) : Ordre de création des tables

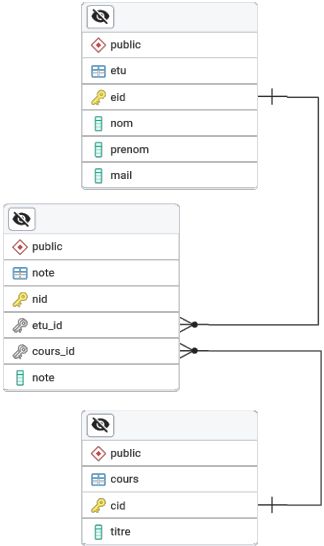


Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval ✓
- Cours, Eval, Etu ✗

```
CREATE TABLE etu(
    ...
); -- plusieurs requête -> ';'
CREATE TABLE cours(
    ...
);
CREATE TABLE eval(
    ...
);
```

Structured Query Language (SQL) : Ordre de création des tables



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✓
- Eval, Cours, Etu ✗
- Eval, Etu, Cours ✗
- Cours, Etu, Eval ✓
- Cours, Eval, Etu ✗

```
CREATE TABLE etu(
    ...
); -- plusieurs requête -> ';'
CREATE TABLE cours(
    ...
);
CREATE TABLE eval(
    ...
);
```

→ Ne pas créer la table si ses dépendances n'existent pas encore !!!

La contrainte DEFAULT

On peut spécifier des valeurs par défaut:

```
nom_colonne TYPE DEFAULT valeur
```

Par exemple si la `note` par défaut est 0 pour la relation **Eval**:

```
CREATE TABLE eval(
    ...
    note FLOAT DEFAULT .0,
    ...
)
```

La contrainte DEFAULT

On peut spécifier des valeurs par défaut:

```
nom_colonne TYPE DEFAULT valeur
```

Par exemple si la `note` par défaut est 0 pour la relation **Eval**:

```
CREATE TABLE eval(
   ...
   note FLOAT DEFAULT .0,
   ...
)
```

Le type SERIAL

Associer un compteur à une colonne

```
nom_colonne SERIAL
```

Par exemple si on souhaite que la valeur de `nid` s'incrémente dans **note**:

```
CREATE TABLE Eval(
   nid SERIAL,
   ...
)
```

⚠ Il s'agit de la méthode pour postgresSQL seulement

Le mots clef ALTER

Il permet de modifier le schéma d'une relation existante, c'est à dire

- Ajouter/modifier/supprimer un champs
- Ajouter/modifier/supprimer une contrainte
- etc...

On commencera toujours par

```
ALTER TABLE nom_de_la_table [mon_operation]
```

Structured Query Language (SQL) : Modification avec ALTER (exemples)

Suppression d'une colonne

```
ALTER TABLE nom_relation
DROP COLUMN nom_colonne
```

Modification du type d'une colonne

```
ALTER TABLE nom_relation
ALTER COLUMN nom_colonne
TYPE type_donnees
```

Ajouter une contrainte de clef primaire

```
ALTER TABLE nom_relation
ADD CONSTRAINT pk_relation
PRIMARY KEY (colonne)
```

Ajout d'une contrainte sur la valeur d'une colonne

```
ALTER TABLE nom_relation
ADD CHECK (nom_colonne>10);
```

On peut supprimer une table avec le mot clef **DROP** :

```
DROP TABLE nom_table
```

Si on considère les relations précédentes que se passe t-il si j'execute le code suivant :

```
DROP TABLE etu
```

On peut supprimer une table avec le mot clef **DROP** :

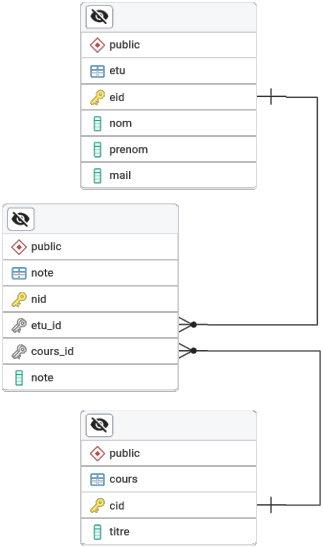
```
DROP TABLE nom_table
```

Si on considère les relations précédentes que se passe t-il si j'execute le code suivant :

```
DROP TABLE etu
```

```
ERROR: constraint fk_note_etu on table \textbf{Eval} depends on table etu...
```

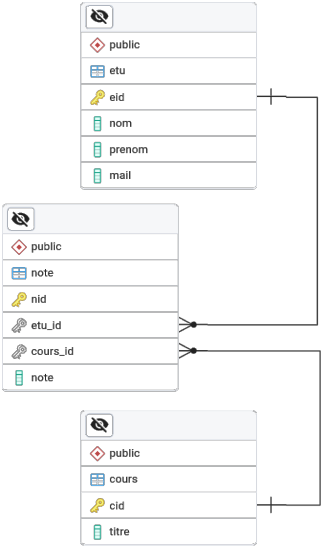
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours

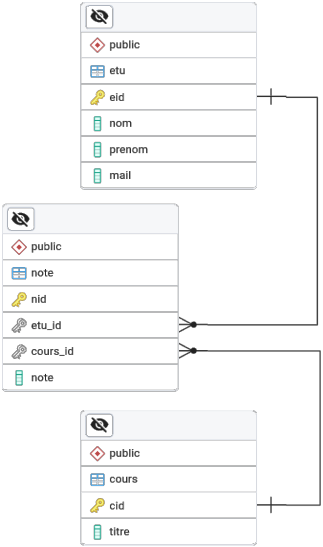
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗

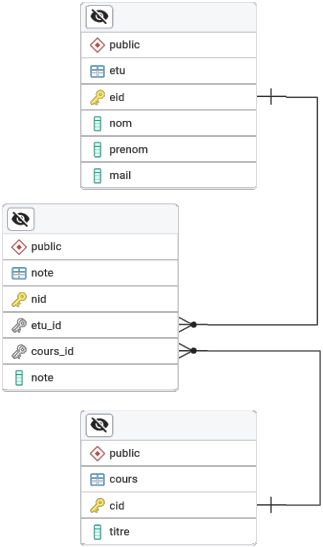
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval

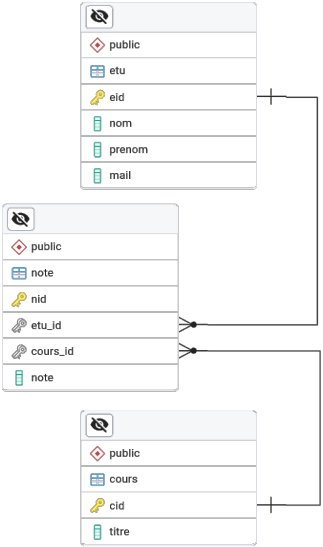
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗

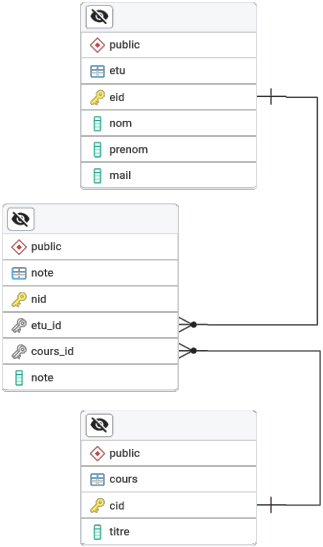
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu

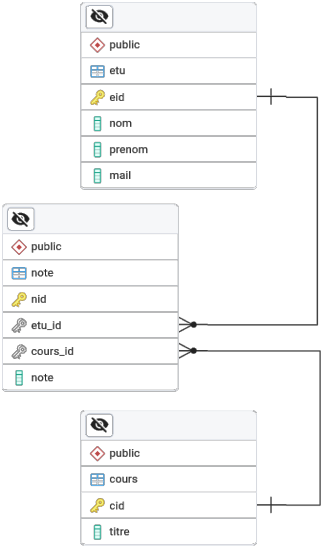
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours
- Etu, Cours, Eval
- Eval, Cours, Etu

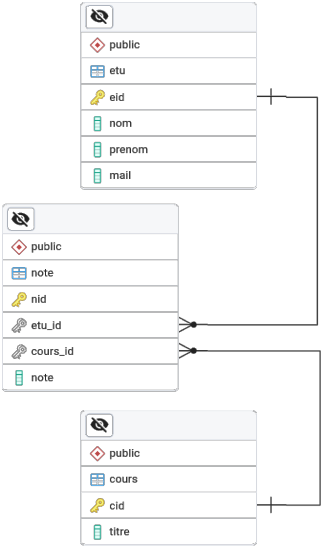
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

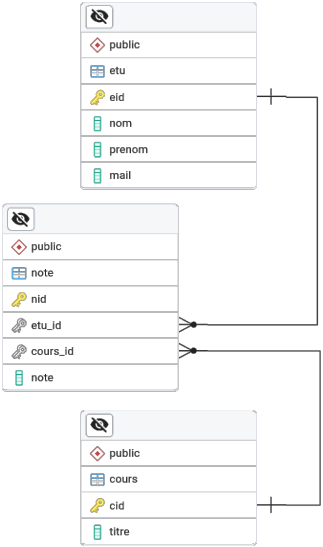
- Etu, Eval, Cours
- Etu, Cours, Eval
- Eval, Cours, Etu
- Eval, Etu, Cours

Structured Query Language (SQL) : Suppression



Dans quel ordre ?

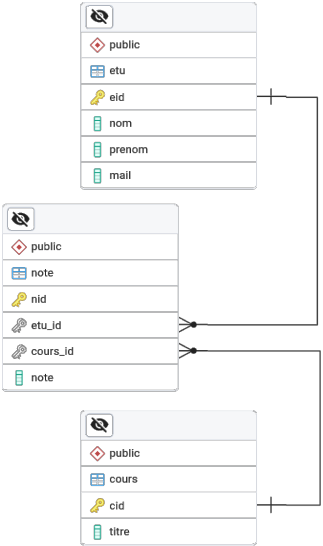
- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓



Dans quel ordre ?

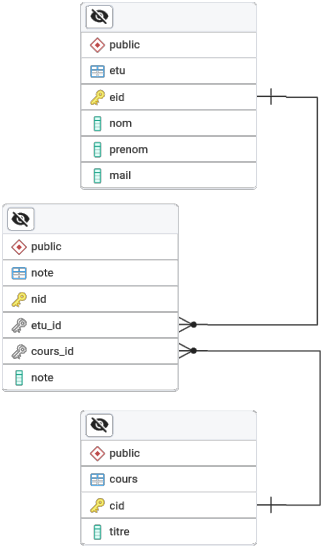
- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval

Structured Query Language (SQL) : Suppression



Dans quel ordre ?

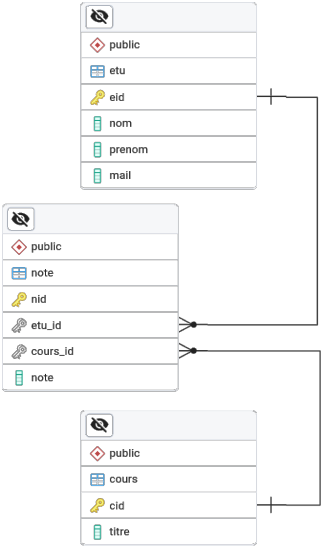
- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval ✗



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval ✗
- Cours, Eval, Etu

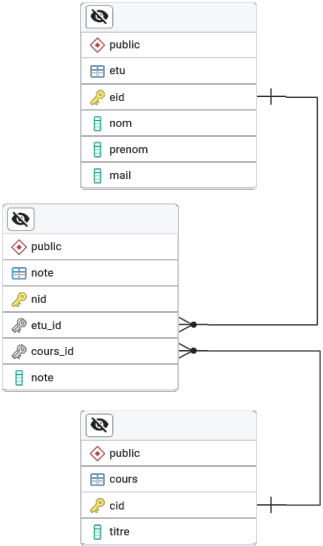
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval ✗
- Cours, Eval, Etu ✗

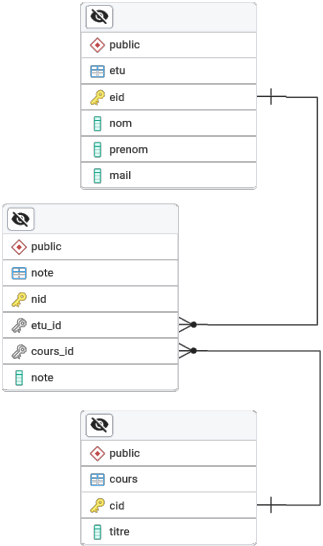
Structured Query Language (SQL) : Suppression



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval ✗
- Cours, Eval, Etu ✗

```
DROP TABLE IF EXISTS eval;
DROP TABLE IF EXISTS cours;
DROP TABLE IF EXISTS etu;
```



Dans quel ordre ?

- Etu, Eval, Cours ✗
- Etu, Cours, Eval ✗
- Eval, Cours, Etu ✓
- Eval, Etu, Cours ✓
- Cours, Etu, Eval ✗
- Cours, Eval, Etu ✗

```
DROP TABLE IF EXISTS eval;
DROP TABLE IF EXISTS cours;
DROP TABLE IF EXISTS etu;
```

→ Détruire les tables dans l'ordre inverse de création
→ Si un champ de la table A référence un champ de la table B alors A doit être détruite avant B

Le langage de définition des données

Le LDD (où DDL), permet de définir la structure des tables :

- Le nom des colonnes
- Le type des colonnes
- Les contraintes
- La création, la modification et la suppression des tables, des colonnes ou des contraintes

Le langage de définition des données

Le LDD (où DDL), permet de définir la structure des tables :

- Le nom des colonnes
- Le type des colonnes
- Les contraintes
- La création, la modification et la suppression des tables, des colonnes ou des contraintes

Comment manipuler les données de la relation ?

Le langage de définition des données

Le LDD (où DDL), permet de définir la structure des tables :

- Le nom des colonnes
- Le type des colonnes
- Les contraintes
- La création, la modification et la suppression des tables, des colonnes ou des contraintes

Comment manipuler les données de la relation ?

↓

Le langage de **Manipulation** des données

Le Language de Manipulation des Données (LDD)

Le LMD

Le langage de manipulation des données (**LMD** ou **DML** en anglais pour data manipulation language) est un langage permettant d'ajouter, supprimer, modifier ou rechercher des données. Une sous-partie du langage SQL définie les opérations du LMD, en particulier via les instructions :

- **INSERT** pour insérer de nouveaux enregistrements
- **SELECT** pour sélectionner certains enregistrements et colonnes
- **DELETE** pour supprimer des enregistrements
- **UPDATE** pour mettre un jour des enregistrements

INSERT INTO	nom_table	(nom_col1, nom_col2)	VALUES	(val_col1, val_col2, ...)
	↓	↓		↓
	dans la table	sur les colonnes		les valeurs

INSERT INTO	nom_table	(nom_col1, nom_col2)	VALUES	(val_col1, val_col2, ...)
	↓	↓		↓
	dans la table	sur les colonnes		les valeurs

Précisions

- Les chaînes de caractères sont délimitées par des guillemets simples
  → `'ma_chaine'`

INSERT INTO	nom_table	(nom_col1, nom_col2)	VALUES	(val_col1, val_col2, ...)
	↓	↓		↓
	dans la table	sur les colonnes		les valeurs

Précisions

- Les chaînes de caractères sont délimitées par des guillemets simples
  → `'ma_chaine'`
- Si l'on conserve l'ordre des colonnes il n'est pas nécessaire de spécifier les noms

INSERT INTO	nom_table	(nom_col1, nom_col2)	VALUES	(val_col1, val_col2, ...)
	↓	↓		↓
	dans la table	sur les colonnes		les valeurs

Précisions

- Les chaînes de caractères sont délimitées par des guillemets simples
  → `'ma_chaine'`
- Si l'on conserve l'ordre des colonnes il n'est pas nécessaire de spécifier les noms
- Certaines colonnes ne sont pas obligatoirement spécifiées
  → pour les colonnes ayant une valeur par défaut ou s'incrémentant automatiquement

INSERT INTO nom_table (nom_col1, nom_col2) **VALUES** (val_col1, val_col2, ...)

↓ dans la table

↓ sur les colonnes

↓ les valeurs

Précisions

- Les chaînes de caractères sont délimitées par des guillemets simples

  → `'ma_chaine'`
- Si l'on conserve l'ordre des colonnes il n'est pas nécessaire de spécifier les noms
- Certaines colonnes ne sont pas obligatoirement spécifiées

  → pour les colonnes ayant une valeur par défaut ou s'incrémentant automatiquement
- Sur certains SGBD on peut ajouter plusieurs enregistrements

  → ...**VALUES** (val_col1_e1, val_col2_e1, ...), (val_col1_e2, val_col2_e2, ...)

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail `ag@mail.com`)

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

```
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

```
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

```
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

```
INSERT INTO etu (prenom, nom, mail) VALUES ('Ali', 'GATOR', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

```
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

```
INSERT INTO etu (prenom, nom, mail) VALUES ('Ali', 'GATOR', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```sql
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

```sql
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

```sql
INSERT INTO etu (prenom, nom, mail) VALUES ('Ali', 'GATOR', 'ag@mail.com')
```

```sql
INSERT INTO etu (eid, prenom, nom, mail) VALUES (1, 'Ali', 'GATOR', 'ag@mail.com')
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

→ Ajout de l'étudiant GATOR Ali (adresse mail **ag@mail.com**)

```
INSERT INTO etu VALUES ( 'Ali', 'GATOR', 'ag@mail.com')
```

❌

```
INSERT INTO etu VALUES (1, 'GATOR', 'Ali', 'ag@mail.com')
```

👎

```
INSERT INTO etu (prenom, nom, mail) VALUES ('Ali', 'GATOR', 'ag@mail.com')
```

👍

```
INSERT INTO etu (eid, prenom, nom, mail) VALUES (1, 'Ali', 'GATOR', 'ag@mail.com')
```

👍

On préférera la 3éme option

Structured Query Language (SQL) : L'instruction INSERT (exemple)

Etu			
eid	nom	prenom	mail
1	Zeblouse	Agathe	az@*****
2	Huai	Odile	oh@*****
3	Peuplu	Jean	jp@*****
4	Hochon	Paul	hp@*****
5	Gator	Ali	ag@*****

Cours	
cid	titre
1	Bases de données avancées
2	Mathématiques
3	Anglais

```
CREATE TABLE eval(
   nid SERIAL,
   etu_id INTEGER,
   cours_id INTEGER,
   note FLOAT DEFAULT .0,
   CONSTRAINT pk_note PRIMARY KEY (nid),
   CONSTRAINT fk_note_etu FOREIGN KEY(etu_id) REFERENCES
   etu(eid),
   CONSTRAINT fk_note_cours FOREIGN KEY(cours_id)
    REFERENCES cours(cid),
   CONSTRAINT u_note_etu_cours UNIQUE(etu_id, cours_id),
   CONSTRAINT c_note CHECK (note >= 0 AND note <= 20),
)
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

Structured Query Language (SQL) : L'instruction INSERT (exemple)

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

nid	etu_id	cours_id	note
1	1	2	0

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

nid	etu_id	cours_id	note
1	1	2	0

```
INSERT INTO eval (etu_id, cours_id, note)
   VALUES (1, 2, 18);
```

Que se passe t-il ?

Structured Query Language (SQL) : L'instruction INSERT (exemple)

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

nid	etu_id	cours_id	note
1	1	2	0

```
INSERT INTO eval (etu_id, cours_id, note)
   VALUES (1, 2, 18);
```

Que se passe t-il ?

```
ERROR: Key (etu_id, cours_id)=(1, 2) already
exists.duplicate key value violates unique
constraint "u_note_etu_cours"
```

Structured Query Language (SQL) : L'instruction INSERT (exemple)

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

nid	etu_id	cours_id	note
1	1	2	0

```
INSERT INTO eval (etu_id, cours_id, note)
   VALUES (1, 2, 18);
```

Que se passe t-il ?

```
ERROR: Key (etu_id, cours_id)=(1, 2) already
exists.duplicate key value violates unique
constraint "u_note_etu_cours"
```

```
INSERT INTO Eval (etu_id, cours_id, note)
VALUES (1, 7, 15);
```

Que se passe t-il ?

Structured Query Language (SQL) : L'instruction INSERT (exemple)

```
INSERT INTO eval (etu_id, cours_id)
   VALUES (1, 2);
```

Quelle est la valeur de l'enregistrement ajouté ?

nid	etu_id	cours_id	note
1	1	2	0

```
INSERT INTO eval (etu_id, cours_id, note)
   VALUES (1, 2, 18);
```

Que se passe t-il ?

```
ERROR: Key (etu_id, cours_id)=(1, 2) already
exists.duplicate key value violates unique
constraint "u_note_etu_cours"
```

```
INSERT INTO Eval (etu_id, cours_id, note)
VALUES (1, 7, 15);
```

Que se passe t-il ?

```
ERROR: Key (cours_id)=(7) is not present in table
"cours".insert or update on table "eval" violates
foreign key constraint "fk_note_cours"
```

SELECT	colonne1, colonne2, ...	FROM	nom_table	WHERE	condition
↓	↓	↓	↓	↓	↓
Sélection	des colonnes 1 et 2	sur	la table	où	une condition

→ `SELECT` pour informer que l'on souhaite "extraire" des enregistrements/colonnes
→ `FROM` on spécifie la ou les relations dans lesquelles se trouvent les enregistrements
→ `WHERE [condition]` on filtre les enregistrements à partir des valeurs d'un ou des champs

On considère les relations précédentes

Sélectionner le champ `note` de la table **eval** →

On considère les relations précédentes

Sélectionner le champ `note` de la table **eval** → `SELECT note From eval;`

Structured Query Language (SQL) : Sélection des champs

On considère les relations précédentes

Sélectionner le champ `note` de la table **eval** →

```
SELECT note From eval;
```

sélectionner le champ `note` et `nid` de la table **eval** →

Structured Query Language (SQL) : Sélection des champs

On considère les relations précédentes

Sélectionner le champ `note` de la table **eval** →

```
SELECT note From eval;
```

sélectionner le champ `note` et `nid` de la table **eval** →

```
SELECT nid, note From eval;
```

On considère les relations précédentes

Sélectionner le champ `note` de la table **eval** →

```
SELECT note From eval;
```

sélectionner le champ `note` et `nid` de la table **eval** →

```
SELECT nid, note From eval;
```

Sélectionner tous les champs de la table **eval** →

Structured Query Language (SQL) : Sélection des champs

On considère les relations précédentes

Sélectionner le champ **note** de la table **eval** →

```
SELECT note From eval;
```

sélectionner le champ **note** et **nid** de la table **eval** →

```
SELECT nid, note From eval;
```

Sélectionner tous les champs de la table **eval** →

```
SELECT * From eval;
```

Si **SELECT** $*$ **FROM** cours retourne la relation suivante :

cid	titre
1	Bases de données avancées
2	Mathématiques
3	Anglais

$\longrightarrow$

Comment faire pour obtenir le résultat ci-dessous ?

Numero	Intitule_du_cours
1	Bases de données avancées
2	Mathématiques
3	Anglais

Si **SELECT** $*$ **FROM** cours retourne la relation suivante :

cid	titre
1	Bases de données avancées
2	Mathématiques
3	Anglais

$\longrightarrow$

Comment faire pour obtenir le résultat ci-dessous ?

Numero	Intitule_du_cours
1	Bases de données avancées
2	Mathématiques
3	Anglais

Utilisation d'un alias (renommage dans le retour de la sélection)

```
SELECT cid AS Numero, titre AS Intitule_du_cours FROM cours
```

⚠ Les noms des colonnes ne sont pas modifiés dans la relation

Comment sélectionner conditionnellement à des valeurs ?

WHERE colonne_ou_valeur **OPÉRATEUR** colonne_ou_valeur_ou_sous_requête

Comment sélectionner conditionnellement à des valeurs ?

WHERE colonne_ou_valeur **OPÉRATEUR** colonne_ou_valeur_ou_sous_requête

Opérateurs boléens

$\rightarrow$ Les opérateurs $<, >, \leq, \geq, =, != $

Comment sélectionner conditionnellement à des valeurs ?

WHERE colonne_ou_valeur OPÉRATEUR colonne_ou_valeur_ou_sous_requête

Opérateurs boléens

→ Les opérateurs $<, >, \leq, \geq, =, !=$
→ L'opérateur `IN` par exemple `col1 IN (`$val_1, val_2, ..., val_n$`)`

Comment sélectionner conditionnellement à des valeurs ?

WHERE colonne_ou_valeur OPÉRATEUR colonne_ou_valeur_ou_sous_requête

Opérateurs boléens

→ Les opérateurs $<, >, \leq, \geq, =, !=$
→ L'opérateur `IN` par exemple `col1 IN (`$val_1, val_2, ..., val_n$`)`
→ Les opérateurs de comparaison de chaînes de caractères (LIKE)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15

→

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator"

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator" →

```
SELECT prenom FROM etu
   WHERE nom = 'Gator'
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator" →

```
SELECT prenom FROM etu
   WHERE nom = 'Gator'
```

Sélectionner les étudiants dont le nom commence par la lettre "H"

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator" →

```
SELECT prenom FROM etu
   WHERE nom = 'Gator'
```

Sélectionner les étudiants dont le nom commence par la lettre "H" →

```
SELECT prenom FROM etu
   WHERE nom LIKE 'H%'
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator" →

```
SELECT prenom FROM etu
   WHERE nom = 'Gator'
```

Sélectionner les étudiants dont le nom commence par la lettre "H" →

```
SELECT prenom FROM etu
   WHERE nom LIKE 'H%'
```

Sélectionner les `eid` des étudiants avec le prénom Agathe ou Odile

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où l'étudiant a eu une note supérieure à 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note > 15;
```

Sélectionner le prénom de l'étudiant de nom "Gator" →

```
SELECT prenom FROM etu
   WHERE nom = 'Gator'
```

Sélectionner les étudiants dont le nom commence par la lettre "H" →

```
SELECT prenom FROM etu
   WHERE nom LIKE 'H%'
```

Sélectionner les `eid` des étudiants avec le prénom Agathe ou Odile →

```
SELECT eid FROM etu
   WHERE prenom IN ('Agathe', 'Odile')
```

Structured Query Language (SQL) : Conjonction et Disjonction

Comment sélectionner selon plusieurs critères ?

→ Sélectionner les étudiants à partir de leurs nom et prénom

→ Les notes entre deux valeurs

Comment sélectionner selon plusieurs critères ?

→ Sélectionner les étudiants à partir de leurs nom et prénom

→ Les notes entre deux valeurs

Les opérateurs OR et AND

On peut cumuler les conditions en utilisant les opérateurs binaires “ou” et “et” :

```
... WHERE (condition1 AND condition2) OR condition3
```

Comment sélectionner selon plusieurs critères ?

→ Sélectionner les étudiants à partir de leurs nom et prénom
→ Les notes entre deux valeurs

Les opérateurs OR et AND

On peut cumuler les conditions en utilisant les opérateurs binaires “ou” et “et” :

```
... WHERE (condition1 AND condition2) OR condition3
```

L’Opérateur NOT

On peut utiliser la négation logique :

```
... WHERE NOT condition
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15 →

```sql
SELECT etu_id, cours_id FROM eval
   WHERE note >= 10 AND note <= 15;
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15 →

```sql
SELECT etu_id, cours_id FROM eval
   WHERE note >= 10 AND note <= 15;
```

Sélectionner les identifiants des étudiants “Agathe Zeblouse” et “Odile Huai”

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note >= 10 AND note <= 15;
```

Sélectionner les identifiants des étudiants “Agathe Zeblouse” et “Odile Huai” →

```
SELECT eid FROM etu
   WHERE
   (prenom='Agathe' AND nom='Zeblouse')
   OR
   (prenom='Odile' AND nom='Huai')
```

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note >= 10 AND note <= 15;
```

Sélectionner les identifiants des étudiants “Agathe Zeblouse” et “Odile Huai” →

```
SELECT eid FROM etu
   WHERE
   (prenom='Agathe' AND nom='Zeblouse')
   OR
   (prenom='Odile' AND nom='Huai')
```

Sélectionner les prénoms des étudiants n’ayant pas de “e” (minuscule) dans leurs nom de famille

Structured Query Language (SQL) : Les conditions (exemples)

Sélectionner les étudiants et les cours où la note est comprise entre 10 et 15 →

```
SELECT etu_id, cours_id FROM eval
   WHERE note >= 10 AND note <= 15;
```

Sélectionner les identifiants des étudiants “Agathe Zeblouse” et “Odile Huai” →

```
SELECT eid FROM etu
   WHERE
   (prenom='Agathe' AND nom='Zeblouse')
   OR
   (prenom='Odile' AND nom='Huai')
```

Sélectionner les prénoms des étudiants n’ayant pas de “e” (minuscule) dans leurs nom de famille →

```
SELECT prenom FROM etu
   WHERE NOT (nom LIKE '%e%')
```

Agrégats

On peut utiliser des agrégats sur les colonnes en utilisant la syntaxe suivante:

```
SELECT FONCTION_AGG(colonne) FROM ma\_table WHERE condition
```

Il existe plusieurs de ces fonctions :

- **MAX**, **MIN** la valeur maximum, la valeur minimum d'une colonne
- **AVG** la valeur moyenne d'une colonne
- **COUNT** le nombre d'enregistrements

Agrégats

On peut utiliser des agrégats sur les colonnes en utilisant la syntaxe suivante:

```
SELECT FONCTION_AGG(colonne) FROM ma\_table WHERE condition
```

Il existe plusieurs de ces fonctions :

- **MAX**, **MIN** la valeur maximum, la valeur minimum d'une colonne
- **AVG** la valeur moyenne d'une colonne
- **COUNT** le nombre d'enregistrements

Un exemple :

Le nombre de notes supérieures à 10

Agrégats

On peut utiliser des agrégats sur les colonnes en utilisant la syntaxe suivante:

```
SELECT FONCTION_AGG(colonne) FROM ma\_table WHERE condition
```

Il existe plusieurs de ces fonctions :

- **MAX**, **MIN** la valeur maximum, la valeur minimum d'une colonne
- **AVG** la valeur moyenne d'une colonne
- **COUNT** le nombre d'enregistrements

Un exemple :

Le nombre de notes supérieures à 10 $\rightarrow$

```
SELECT COUNT(note) AS nb_note FROM eval
   WHERE note > 10
```

Imaginons que nous souhaitions afficher la moyenne de chaque étudiant avec leur identifiant:

Instance de la relation eval

nid	etu_id	cours_id	note
1	1	2	0
4	3	2	15
5	3	3	6
7	1	1	18

Imaginons que nous souhaitions afficher la moyenne de chaque étudiant avec leur identifiant:

Instance de la relation eval

nid	etu_id	cours_id	note
1	1	2	0
4	3	2	15
5	3	3	6
7	1	1	18

→

Résultat souhaité !

etu_id	moyenne
3	10.5
1	9

```
SELECT etu_id, AVG(note) AS moyenne
   FROM eval
```

Imaginons que nous souhaitions afficher la moyenne de chaque étudiant avec leur identifiant:

Instance de la relation eval

nid	etu_id	cours_id	note
1	1	2	0
4	3	2	15
5	3	3	6
7	1	1	18

→

Résultat souhaité !

etu_id	moyenne
3	10.5
1	9

```
SELECT etu_id, AVG(note) AS moyenne
   FROM eval
```

→

```
ERROR: column "eval.etu_id" must
appear in the GROUP BY clause or
be used in an aggregate function
```

Imaginons que nous souhaitions afficher la moyenne de chaque étudiant avec leur identifiant:

Instance de la relation eval

nid	etu_id	cours_id	note
1	1	2	0
4	3	2	15
5	3	3	6
7	1	1	18

→

Résultat souhaité !

etu_id	moyenne
3	10.5
1	9

```
SELECT etu_id, AVG(note) AS moyenne
   FROM eval
```

→

```
ERROR: column "eval.etu_id" must
appear in the GROUP BY clause or
be used in an aggregate function
```

Il faut grouper par identifiant (mais aussi pour toutes les colonnes sur lesquelles aucun agrégat n'est appliqué)

On va utiliser le mot clef GROUP BY

On va utiliser le mot clef GROUP BY

```
SELECT etu_id, AVG(note) AS moyenne
  FROM eval GROUP BY etu_id
```

→

etu_id	moyenne
3	10.5
1	9

On peux aussi trier le résultat ou bien limiter le nombre de résultats retournés :

- ORDER BY permettant de trier (**ASC** pour un trie croissant et **DESC** pour un trie décroissant)
- LIMIT permettant de limiter le nombre d'enregistrements retournés

Sélectionnons les deux meilleures moyennes des étudiants

On va utiliser le mot clef GROUP BY

```
SELECT etu_id, AVG(note) AS moyenne
  FROM eval GROUP BY etu_id
```

→ ✅

etu_id	moyenne
3	10.5
1	9

On peux aussi trier le résultat ou bien limiter le nombre de résultats retournés :

- ORDER BY permettant de trier (**ASC** pour un trie croissant et **DESC** pour un trie décroissant)
- LIMIT permettant de limiter le nombre d'enregistrements retournés

Sélectionnons les deux meilleures moyennes des étudiants

```
SELECT etu_id, AVG(note) AS moyenne
  FROM eval GROUP BY etu_id
  ORDER BY moyenne DESC LIMIT 2
```

→ ✅

etu_id	moyenne
1	9
3	10.5

Sous requête

On peut définir des sous-requêtes pour les conditions (on peut donc utiliser plusieurs tables pour la sélection)

```
SELECT * FROM ma_table WHERE ma_colonne OPERATEUR (SELECT ...)
```

Un exemple :

Le **nom** des étudiants ayant une note supérieure à 17

Sous requête

On peut définir des sous-requêtes pour les conditions (on peut donc utiliser plusieurs tables pour la sélection)

```
SELECT * FROM ma_table WHERE ma_colonne OPERATEUR (SELECT ...)
```

Un exemple :

Le **nom** des étudiants ayant une note supérieure à 17 →

```
SELECT prenom, nom FROM etu
WHERE eid IN
   (
      SELECT etu_id FROM eval
         WHERE note > 17
   )
```

La commande HAVING

Quelquefois nous souhaitons sélectionner à partir d'une condition dépendant d'un agrégat

```
SELECT * FROM ma_table WHERE HAVING FONCTION_AGG(colonne) OPERATEUR valeur;
```

La commande HAVING

Quelquefois nous souhaitons sélectionner à partir d'une condition dépendant d'un agrégat

```
SELECT * FROM ma_table WHERE HAVING FONCTION_AGG(colonne) OPERATEUR valeur;
```

Un exemple :

Les moyennes des étudiants ayant au moins deux notes →

La commande HAVING

Quelquefois nous souhaitons sélectionner à partir d'une condition dépendant d'un agrégat

```
SELECT * FROM ma_table WHERE HAVING FONCTION_AGG(colonne) OPERATEUR valeur;
```

Un exemple :

Les moyennes des étudiants ayant au moins deux notes

→

```
SELECT AVG(note) FROM eval
GROUP BY etu_id HAVING COUNT(note) >= 2
```

Structured Query Language (SQL) : Exercices (au tableau)

En considérant les relations précédentes :

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Question 1 :
Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

Question 2 :
Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

Question 3 :
Retrouver les noms et prénoms des utilisateurs ayant une note supérieure à 10 dans le cours intitulé "Anglais"

Question 4 :
Sélectionner les deux noms de cours avec les meilleures moyennes

Question 1 :
Afficher la moyenne obtenue sur les cours d'identifiant
1 et 3 si celles-ci sont supérieures à 10

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

```
SELECT etu_id FROM eval
   GROUP BY etu_id
   HAVING AVG(note) > (
      SELECT AVG(note) FROM eval);
```

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

```
SELECT etu_id FROM eval
   GROUP BY etu_id
   HAVING AVG(note) > (
      SELECT AVG(note) FROM eval);
```

Question 3 :

Retrouver les noms et prénoms des utilisateurs ayant une note supérieure à 10 dans le cours intitulé “Anglais”

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

```
SELECT etu_id FROM eval
   GROUP BY etu_id
   HAVING AVG(note) > (
      SELECT AVG(note) FROM eval);
```

Question 3 :

Retrouver les noms et prénoms des utilisateurs ayant une note supérieure à 10 dans le cours intitulé "Anglais"

```
SELECT nom, prenom FROM etu
WHERE eid IN ( SELECT etu_id FROM eval
         WHERE cours_id = ( SELECT cid FROM cours
            WHERE titre = 'Anglais'
         ) AND note > 10 )
```

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

```
SELECT etu_id FROM eval
   GROUP BY etu_id
   HAVING AVG(note) > (
      SELECT AVG(note) FROM eval);
```

Question 3 :

Retrouver les noms et prénoms des utilisateurs ayant une note supérieure à 10 dans le cours intitulé "Anglais"

```
SELECT nom, prenom FROM etu
WHERE eid IN ( SELECT etu_id FROM eval
         WHERE cours_id = ( SELECT cid FROM cours
            WHERE titre = 'Anglais'
         ) AND note > 10 )
```

Question 4 :

Sélectionner les deux noms de cours avec les meilleures moyennes

Question 1 :

Afficher la moyenne obtenue sur les cours d'identifiant 1 et 3 si celles-ci sont supérieures à 10

```
SELECT AVG(note) FROM eval
   WHERE (cours_id=1 OR cours_id=3)
   GROUP BY cours_id
   HAVING AVG(note) > 10
```

Question 2 :

Retrouver les identifiants des étudiants ayant une moyenne supérieure à la moyenne globale

```
SELECT etu_id FROM eval
   GROUP BY etu_id
   HAVING AVG(note) > (
      SELECT AVG(note) FROM eval);
```

Question 3 :

Retrouver les noms et prénoms des utilisateurs ayant une note supérieure à 10 dans le cours intitulé "Anglais"

```
SELECT nom, prenom FROM etu
WHERE eid IN ( SELECT etu_id FROM eval
         WHERE cours_id = ( SELECT cid FROM cours
            WHERE titre = 'Anglais'
         ) AND note > 10 )
```

Question 4 :

Sélectionner les deux noms de cours avec les meilleures moyennes

```
SELECT titre FROM cours
   WHERE cid IN (SELECT cours_id FROM
   (SELECT cours_id, AVG(note) as moyenne
      GROUP BY cours_id)
   ORDER BY moyenne DESC LIMIT 2)
```

Structured Query Language (SQL) : ALL et ANY

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ALL(SELECT y FROM ...)
```

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ALL(SELECT y FROM ...)
```

La sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ALL(SELECT y FROM ...)
```

La sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 AND x OPERATOR y_2 AND ... AND x OPERATOR y_3
```

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ALL(SELECT y FROM ...)
```

La sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 AND x OPERATOR y_2 AND ... AND x OPERATOR y_3
```

Exemple : Quels sont les étudiants dont la note minimale est différente de toutes les notes de l'étudiant 1 ?

La commande ALL permet de comparer une valeur à un ensemble de valeurs retourné par une sous-requête. La condition doit être vérifier pour toutes les valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ALL(SELECT y FROM ...)
```

La sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 AND x OPERATOR y_2 AND ... AND x OPERATOR y_3
```

Exemple : Quels sont les étudiants dont la note minimale est différente de toutes les notes de l'étudiant 1 ?

```
SELECT etu_id from eval GROUP BY etu_id HAVING MIN(note) != ALL(
   SELECT note FROM eval WHERE etu_id = 1 )
```

Structured Query Language (SQL) : ANY

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

Structured Query Language (SQL) : ANY

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ANY(SELECT y ...)
```

Structured Query Language (SQL) : ANY

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ANY(SELECT y ...)
```

Et que la sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ANY(SELECT y ...)
```

Et que la sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 OR x OPERATOR y_2 OR ... OR x OPERATOR y_3
```

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ANY(SELECT y ...)
```

Et que la sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 OR x OPERATOR y_2 OR ... OR x OPERATOR y_3
```

Exemple : Quels sont les étudiants dont la note minimale correspond au moins à une note de l'étudiant 1 ?

La commande ANY permet de comparer une valeur à un ensemble de valeurs retournées par une sous-requête. La condition doit être vérifier par au moins une des valeurs retournées par la sous-requête.

```
... WHERE x OPERATOR ANY(SELECT y ...)
```

Et que la sous requête retourne y_1,y_2,..., y_n, alors la requête est équivalente à

```
... WHERE x OPERATOR y_1 OR x OPERATOR y_2 OR ... OR x OPERATOR y_3
```

Exemple : Quels sont les étudiants dont la note minimale correspond au moins à une note de l'étudiant 1 ?

```
SELECT etu_id from eval GROUP BY etu_id HAVING MIN(note) == ANY(
      SELECT note FROM eval WHERE etu_id = 1)
```

Le résultat d'une requête définit un ensemble (un sous ensemble d'une relation) → Les opérations ensemblistes permettent de faire des opérations entres différentes relations (de même schéma)

Un exemple
Soit deux ensembles A et B :

- A contient les notes des étudiants ayant au moins une note en dessous de 7
- B contient les notes des étudiants ayant une moyenne supérieur à 10

Quels sont les étudiants ayant une moyenne supérieure à 10 et au moins une note inférieure à 7

→ Intersection des deux ensembles
Il existe une autre solution en utilisant une conjonction

Les opérateurs ensemblistes

- L'union de deux résultats

```
SELECT * FROM ... UNION SELECT * FROM
```

- L'intersection de deux résultats

```
SELECT * FROM ... INTERSECT SELECT * FROM
```

- La Différence (différent mots clefs selon les SGBDs)

```
SELECT * FROM ... EXCEPT SELECT * FROM
```

- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Proposez une solution faisant intervenir un opérateur ensembliste

Sélectionner les étudiants (identifiants) ayant une moyenne supérieure à 10 mais au moins une note inférieur à 7

- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Proposez une solution faisant intervenir un opérateur ensembliste

Sélectionner les étudiants (identifiants) ayant une moyenne supérieure à 10 mais au moins une note inférieur à 7

```
SELECT etu_id FROM eval GROUP BY etu_id HAVING AVG(note) >= 10
INTERSECTION
SELECT etu_id FROM eval GROUP BY etu_id HAVING MIN(note) < 7
```

- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Proposez une solution faisant intervenir un opérateur ensembliste

Sélectionner les étudiants (identifiants) ayant une moyenne supérieure à 10 mais au moins une note inférieur à 7

```
SELECT etu_id FROM eval GROUP BY etu_id HAVING AVG(note) >= 10
INTERSECTION
SELECT etu_id FROM eval GROUP BY etu_id HAVING MIN(note) < 7
```

Sélectionner les étudiants (identifiants) ayant une moyenne supérieur à 10 ou ayant au moins une note supérieure à 15

- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Proposez une solution faisant intervenir un opérateur ensembliste

Sélectionner les étudiants (identifiants) ayant une moyenne supérieure à 10 mais au moins une note inférieur à 7

```
SELECT etu_id FROM eval GROUP BY etu_id HAVING AVG(note) >= 10
INTERSECTION
SELECT etu_id FROM eval GROUP BY etu_id HAVING MIN(note) < 7
```

Sélectionner les étudiants (identifiants) ayant une moyenne supérieur à 10 ou ayant au moins une note supérieure à 15

```
SELECT etu_id FROM eval GROUP BY etu_id HAVING AVG(note) >= 10
UNION
SELECT etu_id FROM eval GROUP BY etu_id HAVING MAX(note) > 15
```

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

En considérant le schéma précédent on aimerait obtenir une relation comprenant le nom, le prénom, la note et le nom du cours pour chaque étudiant ?

prenom	nom	titre	note
Agathe	Zeblouse	Mathématiques	0
Agathe	Zeblouse	Bases de données avancées	18
Agathe	Zelouse	Anglais	15
Jean	Peuplu	Mathématiques	15
Jean	Peuplu	Anglais	6
Odile	Huai	Bases de données avancées	5
Paul	Hochon	Bases de données avancées	5

Est-il possible avec ce que nous avons déjà vu d'obtenir cette relation ?

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

En considérant le schéma précédent on aimerait obtenir une relation comprenant le nom, le prénom, la note et le nom du cours pour chaque étudiant ?

prenom	nom	titre	note
Agathe	Zeblouse	Mathématiques	0
Agathe	Zeblouse	Bases de données avancées	18
Agathe	Zelouse	Anglais	15
Jean	Peuplu	Mathématiques	15
Jean	Peuplu	Anglais	6
Odile	Huai	Bases de données avancées	5
Paul	Hochon	Bases de données avancées	5

Est-il possible avec ce que nous avons déjà vu d'obtenir cette relation ?→ ⚠ NON !

Le prduit cartésien :

Le produit cartésien entre deux ensemble A et B est noté $A \times B$. Si $A = \{a_1, a_2\}$ et $B = \{b_1, b_2\}$ alors :

Le prduit cartésien :

Le produit cartésien entre deux ensemble A et B est noté $A \times B$. Si $A = \{a_1, a_2\}$ et $B = \{b_1, b_2\}$ alors :

$$A \times B = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$$

C'est à dire l'ensemble des couples !!

Le prduit cartésien :

Le produit cartésien entre deux ensemble A et B est noté $A \times B$. Si $A = \{a_1, a_2\}$ et $B = \{b_1, b_2\}$ alors :

$$A \times B = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$$

C'est à dire l'ensemble des couples !!

$\rightarrow$ En SQL c'est pareil !!! Si A et B sont des relations, le produit cartésien sera l'ensemble des couples d'enregistrements de A et B

Structured Query Language (SQL) : Le produit cartésien

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

$\times$

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

Quel résultat pour le produit cartésien ?

Structured Query Language (SQL) : Le produit cartésien

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

$\times$

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

$\rightarrow$

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
2	Odile	Huai	1	2	0
1	Agathe	Zeblouse	1	1	18
2	Odile	Huai	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	1	3	15

Quel résultat pour le produit cartésien ?

$\rightarrow$ Chaque ligne correspond à la note d'un étudiant ?

Structured Query Language (SQL) : Le produit cartésien

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

$\times$

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

$\rightarrow$

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
2	Odile	Huai	1	2	0
1	Agathe	Zeblouse	1	1	18
2	Odile	Huai	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	1	3	15

Quel résultat pour le produit cartésien ?

$\rightarrow$ Chaque ligne correspond à la note d'un étudiant ? ⚠ NON !

Structured Query Language (SQL) : Le produit cartésien

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

$\times$

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

$\rightarrow$

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
2	Odile	Huai	1	2	0
1	Agathe	Zeblouse	1	1	18
2	Odile	Huai	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	1	3	15

Quel résultat pour le produit cartésien ?

$\rightarrow$ Chaque ligne correspond à la note d'un étudiant ? ⚠ NON !

Structured Query Language (SQL) : Le produit cartésien

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

×

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

→

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
2	Odile	Huai	1	2	0
1	Agathe	Zeblouse	1	1	18
2	Odile	Huai	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	1	3	15

Quel résultat pour le produit cartésien ?

→ Chaque ligne correspond à la note d'un étudiant ? ⚠ NON !

→ Est-il possible avec ce que nous avons déjà vu d'obtenir ce résultat ?

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai

$\times$

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15

$\rightarrow$

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
2	Odile	Huai	1	2	0
1	Agathe	Zeblouse	1	1	18
2	Odile	Huai	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	1	3	15

Quel résultat pour le produit cartésien ?

$\rightarrow$ Chaque ligne correspond à la note d'un étudiant ? ⚠ NON !

$\rightarrow$ Est-il possible avec ce que nous avons déjà vu d'obtenir ce résultat ? OUI !

Produit Cartésien et SQL
En SQL on définie le produit cartésien entre deux relations *A* et *B* comme suit :

```
SELECT ... FROM A, B
```

ou bien,

```
SELECT ... FROM A CROSS JOIN B
```

Il est tout à fait possible de joindre n relations !!!

```
SELECT ... FROM T1 CROSS JOIN T2 CROSS JOIN T3 ... CROSS JOIN TN
```

Comment obtenir une relation comprenant le nom, le prénom, la note et le nom du cours pour chaque étudiant (une solution avec requête imbriqué et une sans)? Calcul du produit cartésien puis filtrage ?

Solution 1

```
SELECT prenom, nom, titre note FROM (SELECT * FROM cours, eval, etu)
WHERE cid=cours_id AND eid=etu_id ;
```

Solution 2

```
SELECT E.prenom, E.nom, C.titre, N.note FROM cours C, eval N, etu E
WHERE C.cid=N.cours_id AND E.eid=E.etu_id ;
```

il s'agit de la jointure

Si on considère la requête suivante

```
SELECT * FROM cours, eval, etu
```

Si le nombre d'enregistrement pour chaque relation est le suivant :

- cours $\rightarrow$ 100
- eval $\rightarrow$ 100,000
- etu $\rightarrow$ 10,000

Quelle est la taille de la relation en sortie de la requête ?

Si on considère la requête suivante

```
SELECT * FROM cours, eval, etu
```

Si le nombre d'enregistrement pour chaque relation est le suivant :

- cours $\rightarrow$ 100
- eval $\rightarrow$ 100,000
- etu $\rightarrow$ 10,000

Quelle est la taille de la relation en sortie de la requête ? $\rightarrow$ ⚠ 10^{11}

La jointure

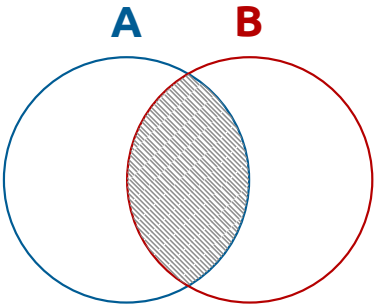
Les jointures en SQL permettent d'associer plusieurs tables dans une même requête.

- CROSS JOIN : Produit cartésien
- INNER JOIN ou JOIN: Association de tables selon les valeurs d'un champs des relations jointes
- LEFT JOIN: Association de tables selon un champs des relations jointes, si les la valeur n'apparaît pas dans la table 'droite' les champs associées à la relation droite sont null
- RIGHT JOIN: Association de tables selon un champs des relations jointes, si les la valeurs n'apparaît pas dans la table 'gauche' les champs associées à la relation gauche sont null

INNER JOIN

INNER JOIN joint deux tables à partir d'un champs. Cette commande retourne les enregistrements lorsqu'il y a au moins une ligne dans chaque colonne qui correspond à la condition.

Quand on parlera de jointure sans préciser nous désignerons la jointure INNER JOIN !!!



Les lignes sont jointes sur les valeurs présentes dans les deux tables

eid	prenom	nom
1	Agathe	Zeblouse
2	Odile	Huai
3	Jean	Peuplu

INNER JOIN (eid=etu_id)

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15
2	1	5

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
1	Agathe	Zeblouse	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	2	1	5

Différentes syntaxes equivalentes

Soit A et B deux relations, on souhaite faire la jointure sur la colonne **id** de A et **aid** de B

```
SELECT * FROM A JOIN B ON A.id = B.aid
```

```
SELECT * FROM A INNER JOIN B ON A.id = B.aid
```

```
SELECT * FROM A , B WHERE A.id = B.aid
```

Si on fait la jointure sur les colonnes **id** de A et **id** de B on peut utiliser NATURAL JOIN

```
SELECT * FROM A NATURAL JOIN B ON A.id = B.aid
```

Structured Query Language (SQL) : Exemple

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants ayant une note supérieure à 10

Retrouver toutes les notes en retournant le nom, le prénom de et le cours

Structured Query Language (SQL) : Exemple

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants ayant une note supérieure à 10

```
SELECT nom, prenom
   FROM etu, eval
   WHERE etu.eid = eval.etu_id AND note > 10
```

Retrouver toutes les notes en retournant le nom, le prénom de et le cours

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants ayant une note supérieure à 10

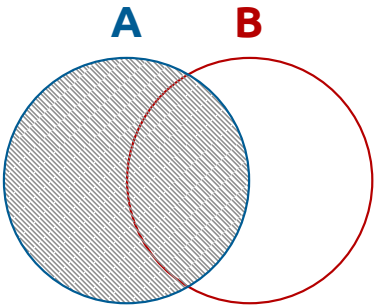
```
SELECT nom, prenom
   FROM etu, eval
   WHERE etu.eid = eval.etu_id AND note > 10
```

Retrouver toutes les notes en retournant le nom, le prénom de et le cours

```
SELECT nom, prenom, cours, note
   FROM etu, eval, cours
   WHERE etu.eid=eval.etu_id
   AND eval.cours_id = cours.cid
```

Jointure gauche

LEFT JOIN joint deux tables à partir d'un champ. La commande permet de lister tous les résultats de la table de gauche même s'il n'y a pas de correspondance dans la deuxième table (les valeurs des colonnes de la table de droite sont nulles lorsque qu'il n'y pas de correspondances).



Les lignes sont jointes sur les valeurs présentes dans les deux tables

1	Agathe	Zeblouse
2	Odile	Huai
3	Jean	Peuplu

LEFT JOIN (eid=etu_id)

etu_id	cours_id	note
1	2	0
1	1	18
1	3	15
2	1	5

eid	prenom	nom	etu_id	cours_id	note
1	Agathe	Zeblouse	1	2	0
1	Agathe	Zeblouse	1	1	18
1	Agathe	Zeblouse	1	3	15
2	Odile	Huai	2	1	5
3	Jean	Peuplu	NULL	NULL	NULL

Différentes syntaxes equivalents

Soit A et B deux relations, on souhaite faire la jointure sur les colonnes `id` de A et `aid` de B

```
SELECT * FROM A LEFT JOIN B ON A.id = B.aid
```

Un exemple

Faire la jointure gauche sur les relations **cours** et **eval** et sélectionnez les colonnes `titre`, `etu_id` et `note` (sur cid=cours_id)

Différentes syntaxes equivalents

Soit A et B deux relations, on souhaite faire la jointure sur les colonnes **id** de A et **aid** de B

```
SELECT * FROM A LEFT JOIN B ON A.id = B.aid
```

Un exemple

Faire la jointure gauche sur les relations **cours** et **eval** et sélectionnez les colonnes **titre**, **etu_id** et **note** (sur cid=cours_id)

```
SELECT titre, etu_id, note FROM cours
   LEFT JOIN eval ON cid=cours_id
```

titre	etu_id	note
Bases de données avancées	1	18
Mathématiques	1	0
Anglais	1	15
Python	NULL	NULL

Structured Query Language (SQL) : Exercices

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants n'ayant pas de notes

Retrouver toutes les notes manquantes en retournant le nom, le prénom de et le cours

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants n'ayant pas de notes

```
SELECT prenom, nom FROM etu AS E
   LEFT JOIN eval AS N ON E.eid=N.etu_id
   WHERE note IS NULL
```

Retrouver toutes les notes manquantes en retournant le nom, le prénom de et le cours

- Etu(*eid*: **serial**, *nom*: **str**, *prenom*: **str**, *mail*: **str**)
- Cours(*cid*: **serial**, *titre*: **str**)
- Eval(*nid*: **serial**, *etu_id*: **int**, *cours_id*: **int**, *note*: **float**)

Obtenir la liste des prénoms et des noms des étudiants n'ayant pas de notes

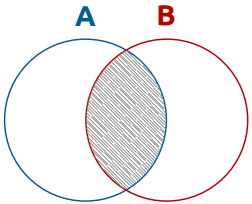
```
SELECT prenom, nom FROM etu AS E
   LEFT JOIN eval AS N ON E.eid=N.etu_id
   WHERE note IS NULL
```

Retrouver toutes les notes manquantes en retournant le nom, le prénom de et le cours et la note

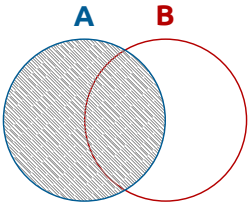
```
SELECT prenom, nom, titre FROM etu E
   CROSS JOIN cours C
   LEFT JOIN eval N ON
      (E.eid=N.etu_id AND N.cours_id=C.cid)
   WHERE note IS NULL
```

Structured Query Language (SQL): Différentes jointures

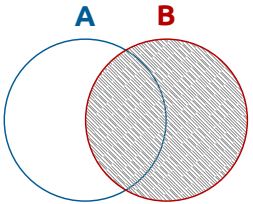
INNER JOIN



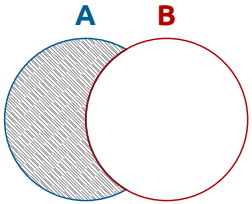
LEFT JOIN



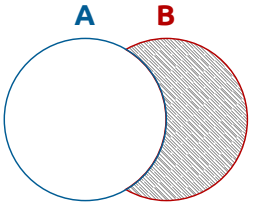
INNER JOIN



LEFT JOIN SANS B



RIGHT JOIN SANS A



Structured Query Language (SQL) : Jointure

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
→	1	Agathe	Zeblouse
	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
→	1	2	0
	1	1	18
	1	3	15
	2	1	5

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
→	1	Agathe	Zeblouse
	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
	1	2	0
→	1	1	18
	1	3	15
	2	1	5

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
→	1	Agathe	Zeblouse
	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
	1	2	0
	1	1	18
→	1	3	15
	2	1	5

Structured Query Language (SQL) : Jointure

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
→	1	Agathe	Zeblouse
	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
	1	2	0
	1	1	18
	1	3	15
→	2	1	5

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
	1	Agathe	Zeblouse
→	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
→	1	2	0
	1	1	18
	1	3	15
	2	1	5

```
SELECT * FROM cours, eval
   WHERE etu.eid=eval.eid
```

eid	prenom	nom	
→	1	Agathe	Zeblouse
	2	Odile	Huai
	3	Jean	Peuplu

⟶

etu_id	cours_id	note	
	1	2	0
→	1	1	18
	1	3	15
	2	1	5

Avec l'algorithme précédent

Combien d'itérations ?

Avec l'algorithme précédent

Combien d'itérations ? $\rightarrow 3 \times 4 = 12$

Avec l'algorithme précédent

Combien d'itérations ? $\rightarrow 3 \times 4 = 12$

Si le nombre de n-uplets dans $n_{etu} = 100$ et $n_{eval} = 1000$
Combien d'itérations ?

Avec l'algorithme précédent

Combien d'itérations ? $\rightarrow 3 \times 4 = 12$

Si le nombre de n-uplets dans $n_{etu} = 100$ et $n_{eval} = 1000$
Combien d'itérations ?

$$100 \times 1.000 = 100.000$$

Si les différentes relations étaient des tableaux (dans un langage quelconques) comment implémenter la requête :

```
SELECT * FROM cours, eval, etu
   WHERE etu.eid=eval.eid
   AND cours.cid=eval.cid
```

→ Boucles imbriquées

Algorithm 1 JOIN

Require: $cours, eval, etu$
  $resultat \leftarrow []$
  for $c \in cours$ **do**
    for $n \in eval$ **do**
      for $e \in etu$ **do**
        if $c.cid = n.cid$ and $e.eid = n.eid$ **then**
          $resultat \leftarrow resultat \cup (c, n, d)$
        end if
      end for
    end for
  end for

Si les différentes relations étaient des tableaux (dans un langage quelconques) comment implémenter la requête :

```
SELECT * FROM cours, eval, etu
   WHERE etu.eid=eval.eid
   AND cours.cid=eval.cid
```

Combien d'itérations ?

Algorithm 2 JOIN

Require: $cours, eval, etu$
  $resultat \leftarrow []$
  for $c \in cours$ **do**
    for $n \in eval$ **do**
      for $e \in etu$ **do**
        if $c.cid = n.cid$ and $e.eid = n.eid$ **then**
          $resultat \leftarrow resultat \cup (c, n, d)$
        end if
      end for
    end for
  end for

Si les différentes relations étaient des tableaux (dans un langage quelconques) comment implémenter la requête :

```
SELECT * FROM cours, eval, etu
   WHERE etu.eid=eval.eid
   AND cours.cid=eval.cid
```

Combien d'itérations ?

$$n_{cours} \times n_{eval} \times n_{etu}$$

Algorithm 3 JOIN

Require: $cours, eval, etu$
  $resultat \leftarrow []$
  for $c \in cours$ **do**
    for $n \in eval$ **do**
      for $e \in etu$ **do**
        if $c.cid = n.cid$ and $e.eid = n.eid$ **then**
          $resultat \leftarrow resultat \cup (c, n, d)$
        end if
      end for
    end for
  end for

Si les différentes relations étaient des tableaux (dans un langage quelconques) comment implémenter la requête :

```
SELECT * FROM cours, eval,
     etu
  WHERE etu.eid=eval.eid
  AND cours.cid=eval.cid
```

Combien d'itérations ?

$$n_{cours} \times n_{eval} \times n_{etu}$$

Algorithm 4 JOIN

Require: $cours, eval, etu$
  $resultat \leftarrow []$
  for $c \in cours$ **do**
    for $n \in eval$ **do**
      for $e \in etu$ **do**
        if $c.cid = n.cid$ and $e.eid = n.eid$ **then**
          $resultat \leftarrow resultat \cup (c, n, d)$
        end if
      end for
    end for
  end for

Il faut optimiser (Partie Optimisation)

La mise à jour

On peut mettre à jour un enregistrement dans la table, en utilisant l'instruction UPDATE:

```
UPDATE nom_table SET nom_colonne_1=a nom_colonne_2=b WHERE condition;
```

Modifier la note en mathématiques (cours_id=2) de Agathe Zeblouse (etu_id=1) à 20

```
UPDATE Eval SET note = 20 WHERE etu_id=1 AND cours_id = 2
```

On peut aussi utiliser des SELECT pour retrouver l'enregistrement

La mise à jour

On peut mettre à jour un enregistrement dans la table, en utilisant l'instruction UPDATE:

```
UPDATE nom_table SET nom_colonne_1=a nom_colonne_2=b WHERE condition;
```

Modifier la note en mathématiques (cours_id=2) de Agathe Zeblouse (etu_id=1) à 20

```
UPDATE Eval SET note = 20 WHERE etu_id=1 AND cours_id = 2
```

On peut aussi utiliser des SELECT pour retrouver l'enregistrement

```
UPDATE Eval SET note = 20
   WHERE etu_id=(SELECT eid FROM etu WHERE nom='Zeblouse' AND prenom='Agathe')
   AND cours_id = (SELECT cid FROM cours WHERE titre='Mathématiques')
```

Mettre à jour plusieurs enregistrements (exemple)

On peut mettre a jour plusieurs enregistrements en utilisant les valeurs des lignes retrouvées

Exemple : Enlever deux points à tous les étudiants

Mettre à jour plusieurs enregistrements (exemple)

On peut mettre a jour plusieurs enregistrements en utilisant les valeurs des lignes retrouvées

Exemple : Enlever deux points à tous les étudiants

```
UPDATE Eval SET note = note-2;
```

nom	prenom	cours	note
Peuplu	Jean	Mathématiques	15
Peuplu	Jean	Anglais	6
Zeblouse	Agathe	Bases de données ...	18
Zeblouse	Agathe	Mathématiques	20

→

nom	prenom	cours	note
Peuplu	Jean	Mathématiques	13
Peuplu	Jean	Anglais	4
Zeblouse	Agathe	Bases de données ...	16
Zeblouse	Agathe	Mathématiques	18

Pour aller plus loin !

Il existe d'autre syntaxe fonctionnalités

- Site de PostgreSQL`https://docs.postgresql.fr/9.6/sql-update.html`
- W3Schools `https://www.w3schools.com/sql/sql_update.asp`

Suppression d'un enregistrement
On peut supprimer un enregistrement dans la table, en utilisant l'instruction DELETE:

```
DELETE FROM nom_table WHERE condition;
```

Exemple : Supprimer les notes du cours Mathématiques

Suppression d'un enregistrement
On peut supprimer un enregistrement dans la table, en utilisant l'instruction DELETE:

```
DELETE FROM nom_table WHERE condition;
```

Exemple : Supprimer les notes du cours Mathématiques

```
DELETE FROM eval WHERE cours_id = (SELECT cid WHERE titre='Mathématiques' )
```

Informations supplémentaires

SELECT DISTINCT

L'instruction SELECT DISTINCT est utilisée pour renvoyer uniquement les enregistrements distincts.

```
SELECT DISTINCT
```

STRING_AGG

La fonction STRING_AGG concatène des chaînes de caractères séparées par un séparateur spécifié.

Ressources

- Documentation postgreSQL `https://www.postgresql.org/docs/16/index.html`
- W3School `https://www.w3schools.com/sql`

Conclusion

Un petit TP

Objectifs

- Utilisation de requêtes SQL ”simples”
- Observer/comprendre les enjeux des SGBDs

Disponible sur `https://cquae-annotation.freeboxos.fr/BDA/`

- identifiant : bda
- mot de passe : 24bda25

Dans la suite

Ce qui a été vue

- Initiation au langage SQL
- Les opérations sur les bases de données relationnelles avec SQL

Comment ces opérations sont implémentées dans les SGBDs ?

Comment garantir l'efficacité des opérations ? $\rightarrow$ Optimisation des SGBDs

Comment garantir l'intégrité des données ? $\rightarrow$ Concurrence (propriétés ACID)