# Mise à niveau Bases de Données Relationnelles : SQL Fondamental avec PostgreSQL

## Formation complète – Niveau M1 Informatique

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

## Table des matières

0.1 Objectif ..... 3
0.2 Prérequis techniques ..... 4
0.3 Objectifs pédagogiques détaillés ..... 5
0.4 Structure du TP ..... 5
0.4.a ◯ Parcours Minimal ..... 5
0.4.b ◯ Parcours Approfondi ..... 6
0.4.c ◯ Parcours Avancé ..... 6
1 BLOC 1 : Modèle relationnel et fondations théoriques ..... 6
1.1 Le modèle relationnel : rappels théoriques ..... 6
1.1.a Définitions formelles ..... 7
1.1.b Propriétés fondamentales ..... 7
1.1.c Clés : primaires, candidates, étrangères ..... 9
1.1.d Contraintes d'intégrité ..... 10
1.2 Implémentation du schéma ..... 11
1.2.a Création du schéma initial ..... 11
1.2.b Analyse détaillée du code ..... 13
1.3 Insertion de données de test ..... 14
1.3.a Visualisation des données insérées ..... 15
1.4 Exercices de compréhension du schéma ..... 17
2 BLOC 2 : Requêtes SELECT — Fondamentaux et filtrage ..... 18
2.1 Anatomie d'une requête SELECT ..... 18
2.2 SELECT simple et projection ..... 19
2.2.a Sélection de toutes les colonnes ..... 19
2.2.b Projection de colonnes spécifiques ..... 20
2.2.c Alias de colonnes ..... 20

2.3 Clause WHERE : filtrage de lignes ..... 21
2.3.a Opérateurs de comparaison ..... 21
2.3.b Opérateurs logiques ..... 22
2.4 ORDER BY : tri des résultats ..... 22
2.5 LIMIT et OFFSET : pagination ..... 24
2.6 Exercices — SELECT et filtrage ..... 25
3 BLOC 3 : Jointures — Reconstruction de l'information ..... 26
3.1 Comprendre le problème : produit cartésien ..... 26
3.2 INNER JOIN : jointure naturelle ..... 28
3.2.a Syntaxe moderne (recommandée) ..... 28
3.2.b Syntaxe ancienne (WHERE) ..... 28
3.3 Jointures multiples ..... 29
3.4 LEFT JOIN : jointure externe gauche ..... 31
3.4.a Problématique ..... 31
3.4.b Solution : LEFT JOIN ..... 31
3.5 Autres types de jointures ..... 32
3.5.a RIGHT JOIN ..... 32
3.5.b FULL OUTER JOIN ..... 33
3.6 Exercices — Jointures ..... 35
4 BLOC 4 : Fonctions d'agrégat et GROUP BY ..... 36
4.1 Fonctions d'agrégat : résumer les données ..... 36
4.1.a Fonctions de base ..... 36
4.2 GROUP BY : agrégation par groupe ..... 37
4.2.a Principe ..... 38
4.2.b Groupement sur clé primaire ..... 39
4.3 HAVING : filtrage après agrégation ..... 40
4.3.a Problématique ..... 40
4.3.b Solution : HAVING ..... 40
4.4 Exemples avancés ..... 41
4.4.a Nombre d'étudiants par cours avec détails ..... 41
4.5 Exercices — Agrégats ..... 43
5 BLOC 5 : Sous-requêtes et requêtes avancées ..... 44
5.1 Sous-requêtes scalaires ..... 44
5.2 Sous-requêtes avec IN ..... 45
5.3 Sous-requêtes corrélées ..... 46
5.4 CTE (Common Table Expressions) ..... 47
5.5 UNION et opérations ensemblistes ..... 47
6 BLOC 6 : Performance et optimisation ..... 48
6.1 Index : accélérer les recherches ..... 48
6.1.a Principe ..... 48

6.1.b Création d'index ..... 49
6.2 EXPLAIN ANALYZE : comprendre l'exécution ..... 49
6.3 Exercices — Optimisation ..... 51
7 BLOC 7 : Conception et normalisation ..... 51
7.1 Anomalies de mise à jour ..... 51
7.1.a Schéma dénormalisé (mauvais) ..... 51
7.2 Formes normales ..... 52
7.3 Exercice de conception ..... 54
8 BLOC 8 : Transactions et isolation ..... 56
8.1 Introduction aux transactions : Garantir la cohérence ..... 56
8.1.a Définition et propriétés ACID ..... 56
8.2 Commandes de base ..... 57
8.2.a BEGIN, COMMIT, ROLLBACK ..... 57
8.3 Exemple pratique : inscription cohérente ..... 58
8.3.a Problème sans transaction ..... 58
8.3.b Solution avec transaction ..... 58
8.4 Savepoints : points de sauvegarde ..... 59
8.5 Niveaux d'isolation : comprendre la concurrence ..... 60
8.5.a Problématique : anomalies de lecture ..... 60
8.5.b Les 4 niveaux d'isolation SQL ..... 62
8.5.c Démonstration des niveaux d'isolation ..... 62
8.6 Verrous (Locks) : contrôle explicite ..... 65
8.6.a Types de verrous ..... 65
8.6.b SELECT FOR UPDATE ..... 65
8.6.c FOR UPDATE vs FOR SHARE ..... 66
8.6.d Deadlocks : interblocages ..... 66
8.7 Patterns de concurrence courants ..... 67
8.7.a Pattern 1 : Compteur incrémental thread-safe ..... 67
8.7.b Pattern 2 : Places limitées (overbooking) ..... 68
8.7.c Pattern 3 : Optimistic Locking (version) ..... 69
8.8 Exercices — Transactions et isolation ..... 71
8.9 Bonnes pratiques et recommandations ..... 74
8.9.a Choix du niveau d'isolation ..... 74
8.9.b Règles d'or ..... 75
8.9.c Monitoring ..... 75
8.10 Conclusion ..... 75

## 0.1 Objectif

Passer de la théorie à la pratique en manipulant un SGBD relationnel professionnel (PostgreSQL).

## 0.2 Prérequis techniques

> ! Configuration requise
>
> Avant de commencer, assurez-vous d'avoir :
>
> 1. **PostgreSQL 14+** installé et démarré
> 2. Une base de données nommée `tp_sql` créée

Soit avec une installation locale de PostgreSQL,

```
# Création de la base (en ligne de commande)
createdb -U postgres tp_sql
```

soit via docker

```
export DB_HOST=...
export DB_PORT=...
export DB_USERNAME=...
export DB_PASSWORD=...
export DB_NAME=...

docker rm -f db >/dev/null 2>&1 || true
docker run --label teaching-demo=true --quiet -d --name db \
    --env POSTGRES_USER=${DB_USERNAME} \
    --env POSTGRES_PASSWORD=${DB_PASSWORD} \
    --env POSTGRES_DB=${DB_NAME} \
    --network=notebooknet \
    --tmpfs /var/lib/postgresql/data \
    --health-cmd "pg_isready -U ${DB_USERNAME} -d ${DB_NAME}" \
    --health-interval 2s \
    --health-timeout 2s \
    --health-retries 10 \
    -p 5432:5432 \
    postgres:17
```

```
20356d181ef568c2257fd37c57f36614f1c0e7b7a27484d3eeed44bc521181b6
⟶ Reset du conteneur postgresql
f168937c477c78c16e6b39f4d022b91dde99e19615771e519f64bbd4c7ef4cbd
⟶ Attente de la disponibilité du conteneur sur le réseau...
ERREUR : La base sur dind ne répond pas.
12:12:52.461  [IJava-executor-0]  WARN    o.a.h.c.p.ResponseProcessCookies  --
Invalid cookie header: "set-cookie: __cf_bm=jWpPEweYMHniJrGyKBZgOqMV5k.etk6cXxnIgobyJ2Q-1793535172.
HSx8i1kMy7.lVhE2NHZlkLmcnWlm4ZMYWaDm.tIVmZU0YprjaX9nZvcJixa2wvP4qdfYS2b0qAVjUq2SztjSnxm_ths7WwTAZ
```

```
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Sun, 01 Nov 2026 12:42:52 GMT". Invalid 'expires' attribute: Sun, 01 Nov
2026 12:42:52 GMT
12:12:52.605  [IJava-executor-0]  WARN    o.a.h.c.p.ResponseProcessCookies  --
Invalid cookie header: "set-cookie: __cf_bm=EjHI0I2yIhaRaN4zPsEudgKw5Th2TIvnGka_rSNrl7I-1793535172.
AzukzO3PfZ14qqaHr1aTCvuzWcXaK8HNwYRRSPovLt6bi2XBUAENM8FhI92omQL_pzmcHoRPYZ_Ek7vCKbLz1C1R1sgcbLdl.
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Sun, 01 Nov 2026 12:42:52 GMT". Invalid 'expires' attribute: Sun, 01 Nov
2026 12:42:52 GMT
12:12:52.661     [BasicRepositoryConnector-repo.maven.apache.org-0-0]     WARN
o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie: __cf_bm=l7K3HDWtSbt1YCTnb4c
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Sun, 01 Nov 2026 12:42:52 GMT". Invalid 'expires' attribute: Sun, 01 Nov
2026 12:42:52 GMT
12:12:56.074     [BasicRepositoryConnector-repo.maven.apache.org-0-1]     WARN
o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie: __cf_bm=LgA_yf_U8MjXav1AI4i
F8iQkFvMTbVqlzzsEYSbK0Zkmt4uOzwchQSHLi2wayFnN8cEpxYrw7qHIPf5xHNg.OuC3uhmvAHMZwBR0DP2MCXrqxbj3ueao
HttpOnly;   SameSite=None;   Secure;   Path=/;   Domain=repo.maven.apache.org;
Expires=Sun, 01 Nov 2026 12:42:52 GMT". Invalid 'expires' attribute: Sun, 01 Nov
2026 12:42:52 GMT
```

## 0.3 Objectifs pédagogiques détaillés

À l'issue de ce TP, vous serez capable de :

1. **Modélisation** : Concevoir un schéma relationnel normalisé
2. **Implémentation** : Créer tables, contraintes et index en SQL
3. **Interrogation** : Écrire des requêtes SELECT complexes avec jointures
4. **Agrégation** : Utiliser GROUP BY, HAVING et les fonctions d'agrégat
5. **Optimisation** : Comprendre les index et analyser les plans d'exécution
6. **Intégrité** : Manipuler les contraintes d'intégrité référentielle

## 0.4 Structure du TP

Le TP est organisé en **parcours progressifs** :

### 0.4.a ◯ Parcours Minimal

Concepts essentiels pour réussir le TP :

- Modèle relationnel : tables, clés, tuples
- Requêtes SELECT de base
- Jointures INNER JOIN
- Agrégats simples (COUNT, AVG, SUM)

**Public cible** : Étudiants découvrant SQL

#### 0.4.b ◯ Parcours Approfondi

Concepts intermédiaires + pratique :

- Tout le parcours minimal
- Clés étrangères et intégrité référentielle
- Jointures LEFT/RIGHT/FULL OUTER
- GROUP BY avec HAVING
- Sous-requêtes corrélées

**Public cible** : Étudiants ayant déjà manipulé SQL

#### 0.4.c ◯ Parcours Avancé

Concepts experts + optimisation :

- Tout le parcours approfondi
- Normalisation (3NF, BCNF)
- Index et performance
- EXPLAIN ANALYZE
- Vues matérialisées
- CTE (Common Table Expressions)

**Public cible** : Étudiants visant l'excellence

---

# 1 BLOC 1 : Modèle relationnel et fondations théoriques

## 1.1 Le modèle relationnel : rappels théoriques

Un Système de Gestion de Base de Données Relationnelle (SGBDR) repose sur le **modèle relationnel** défini par E. F. Codd en 1970. Le modèle relationnel repose sur des concepts mathématiques rigoureux, notamment la théorie des ensembles et la logique relationnelle.

Il est ensuite implémenté dans des systèmes de gestion de bases de données relationnelles (SGBDR) tels que PostgreSQL, MySQL, Oracle, SQL Server, etc. Une **table** est l'implémentation concrète d'une relation dans un SGBDR.

L'**algèbre relationnelle** fournit un ensemble d'opérations pour manipuler les relations (tables) de manière formelle. Le langage **SQL** (Structured Query Language) est une implémentation pratique de ces concepts, permettant aux utilisateurs de définir, manipuler et interroger des bases de données relationnelles.

### 1.1.a Définitions formelles

> **Définition : Relation**
>
> Une **relation** $R$ est un sous-ensemble du produit cartésien de $n$ domaines :
>
> $$R \subseteq D_1 \times D_2 \times ... \times D_n$$
>
> Où :
>
> - Chaque $D_i$ est un **domaine** (ensemble de valeurs possibles)
> - Un élément de $R$ est un **tuple** (ou **n-uplet**)
> - L'ensemble des attributs constitue le **schéma** de la relation

**Exemple concret** :

```
Étudiant(id: INTEGER, nom: VARCHAR, prenom: VARCHAR, date_naissance: DATE)
```

- **Domaines** : $\mathbb{N}$ pour `id`, $\Sigma^*$ pour `nom` et `prenom`, DATE pour `date_naissance`
- **Tuple** : $(1, \text{"Dupont"}, \text{"Alice"}, 1999\text{-}03\text{-}15)$
- **Relation** : Ensemble de tous les tuples respectant le schéma

### 1.1.b Propriétés fondamentales

Une relation possède les propriétés suivantes :

1. **Pas d'ordre entre les tuples** : $\{t_1, t_2\} = \{t_2, t_1\}$
2. **Pas de doublons** : ensemble au sens mathématique
3. **Atomicité** : chaque attribut contient une valeur indivisible
4. **Schéma fixe** : tous les tuples ont les mêmes attributs

Chaque attribut a un **type** (domaine) défini. Les types de PostgreSQL (et du standard SQL) sont :

| Catégorie | Type PostgreSQL | Description | Différence / Particularité |
|---|---|---|---|
| **Numérique** | `SMALLINT` | Entier 2 octets (-32768 à +32767) | Standard SQL. |
| | `INTEGER` | Entier 4 octets | Standard SQL. |
| | `BIGINT` | Entier 8 octets | Standard SQL. |
| | `NUMERIC(p,s)` | Nombre exact haute précision | Recommandé pour la finance (alias `DECIMAL`). |

| Catégorie | Type PostgreSQL | Description | Différence / Particularité |
|---|---|---|---|
| | `REAL` / `DOUBLE PRECISION` | Virgule flottante (4 ou 8 octets) | `DOUBLE PRECISION` est le standard pour le calcul. |
| **Auto-incrément** | `SERIAL` / `BIGSERIAL` | Entier auto-incrémenté | Spécifique PG (crée une `SEQUENCE` en arrière-plan). |
| **Texte** | `VARCHAR(n)` | Chaîne de longueur limitée | Standard SQL. |
| | `CHAR(n)` | Chaîne de longueur fixe | Complète avec des espaces (standard). |
| | `TEXT` | Chaîne de longueur illimitée | **Spécificité PG** : Pas de perte de performance vs VARCHAR. |
| **Temporel** | `DATE` | Date seule | Standard SQL. |
| | `TIME [TZ]` | Heure seule (avec/sans fuseau) | Standard SQL. |
| | `TIMESTAMP [TZ]` | Date + Heure (avec/sans fuseau) | `TIMESTAMPTZ` est recommandé pour l'UTC. |
| | `INTERVAL` | Durée (ex: "1 day 2 hours") | Très puissant dans PostgreSQL. |
| **Logique** | `BOOLEAN` | TRUE, FALSE ou NULL | Accepte aussi "y", "n", "1", "0" en entrée. |
| **Binaire** | `BYTEA` | Données binaires (« Byte Array ») | Remplace le standard `BLOB` dans PostgreSQL. |
| **Structuré** | `JSON` | Texte JSON valide | Vérifie la syntaxe, mais stocke tel quel. |

| Catégorie | Type PostgreSQL | Description | Différence / Particularité |
|---|---|---|---|
| | `JSONB` | JSON binaire optimisé | **Spécificité PG** : Supporte l'indexation (très rapide). |
| **Réseau** | `CIDR` / `INET` | Adresses IP (IPv4/IPv6) | **Spécificité PG** : Valide le format IP nativement. |
| | `MACADDR` | Adresse MAC | Valide le format des cartes réseau. |
| **Géométrique** | `POINT`, `LINE`, `POLYGON` | Coordonnées 2D | Support natif des formes géométriques. |
| **Identifiant** | `UUID` | Universally Unique Identifier | Type natif (plus performant qu'un VARCHAR). |
| **Recherche** | `TSVECTOR` / `TSQUERY` | Full-Text Search | Dédié à la recherche textuelle avancée. |

⚠️ ⚠️ Erreur fréquente

En SQL, une **table** peut contenir des doublons (multiset). Pour respecter le modèle relationnel strict, il faut :

- Définir une **clé primaire** (`PRIMARY KEY`)
- Utiliser **DISTINCT** dans les `SELECT` si nécessaire

#### 1.1.c Clés : primaires, candidates, étrangères

Pour assurer l'**unicité** des tuples dans une relation, on utilise des **clés**.

Définition : Clé candidate

Un ensemble d'attributs $K = \{A_1, A_2, ..., A_k\}$ est une **clé candidate** si elle vérifie :

1. **Unicité** : $\forall t_1, t_2 \in R, t_1[K] = t_2[K] \Rightarrow t_1 = t_2$
2. **Minimalité** : $\nexists K' \subset K$ vérifiant l'unicité

Si la clé est composée de plusieurs attributs, on parle de **clé composée** (composite key).

**Exemple** :

Pour la relation :

```
Personne(num_secu, email, nom, prenom)
```

On a deux clés candidates possibles `{num_secu}` et `{email}`. On peut choisir l'une d'elles comme clé primaire par exemple `{num_secu}`. Pour la seconde il faut s'assurer que l'email est toujours renseigné et unique (contrainte `NOT NULL` + `UNIQUE`).

Pour définir des liens entre relations, on utilise des **clés étrangères**. Une **clé étrangère** est un ensemble d'attributs dans une relation qui référence la clé primaire d'une autre relation.

> **Définition : Clé étrangère**
>
> Soit $R_1$ et $R_2$ deux relations. Un ensemble d'attributs $FK$ de $R_1$ est une **clé étrangère** référençant $R_2$ si :
>
> $$FK \subseteq \text{attributs}(R_1) \text{ et } \forall t \in R_1, t[FK] \in \pi_K(R_2) \cup \{\text{NULL}\}$$
>
> où $K$ est la clé primaire de $R_2$.

Par exemple , dans une relation `Inscription(etudiant_id, cours_id, date_inscription)`, `etudiant_id` peut être une clé étrangère référençant `Etudiant(id)`. La clé étrangère garantit que chaque `etudiant_id` dans `Inscription` correspond à un `id` existant dans `Etudiant`. Elle peut faire partie d'une clé primaire composée.

**Intuition** : Une clé étrangère crée un **lien logique** entre deux tables.

#### 1.1.d Contraintes d'intégrité

Le langage SQL permet de définir des contraintes pour garantir l'intégrité des données.

| Contrainte | Rôle | Exemple SQL |
|---|---|---|
| **PRIMARY KEY** | Unicité + NOT NULL | `id INTEGER PRIMARY KEY` |
| **UNIQUE** | Unicité (NULL autorisé) | `email VARCHAR UNIQUE` |
| **NOT NULL** | Valeur obligatoire | `nom VARCHAR NOT NULL` |

| Contrainte | Rôle | Exemple SQL |
|---|---|---|
| **FOREIGN KEY** | Intégrité référentielle | `FOREIGN KEY (etudiant_id) REFERENCES etudiants(id)` |
| **CHECK** | Validation métier | `CHECK (note ⩾ 0 AND note ⩽ 20)` |

## 1.2 Implémentation du schéma

Il exiset un premier sous ensemble du langage SQL appelé DDL (Data Definition Language) qui permet de définir et modifier la structure des objets de la base de données (tables, contraintes, index, vues, etc.).

### 1.2.a Création du schéma initial

Nous allons modéliser un système de **gestion pédagogique** simple :

- Des **étudiants** suivent des **cours**
- Chaque suivi est associé à une **note**

> 💡 Analyse métier
>
> **Questions à se poser** :
>
> 1. Un étudiant peut-il suivre plusieurs cours ? **Oui** → relation 1:N
> 2. Un cours peut-il avoir plusieurs étudiants ? **Oui** → relation N:M
> 3. La note dépend-elle uniquement du cours ou du couple (étudiant, cours) ? **Du couple** → attribut de la relation N:M
>
> **Conclusion** : Il faut une **table d'association** pour la relation N:M.

```
%%sqlAsTable
//| attr-code: ".sql"

-- Suppression des tables si elles existent (ordre important !)
-- La CASCADE permet de supprimer les dépendances automatiquement
DROP TABLE IF EXISTS inscriptions CASCADE;
DROP TABLE IF EXISTS etudiants CASCADE;
DROP TABLE IF EXISTS cours CASCADE;

-- Table ETUDIANTS
CREATE TABLE etudiants (
    id SERIAL PRIMARY KEY,
    nom VARCHAR(50) NOT NULL,
    prenom VARCHAR(50) NOT NULL,
```

```
    email VARCHAR(100) UNIQUE,
    date_naissance DATE,
    CONSTRAINT email_valide CHECK (email LIKE '%@%.%')
);

-- Table COURS
CREATE TABLE cours (
    id SERIAL PRIMARY KEY,
    code VARCHAR(10) UNIQUE NOT NULL,
    nom VARCHAR(100) NOT NULL,
    credits INTEGER CHECK (credits > 0),
    CONSTRAINT code_format CHECK (code ~ '^[A-Z]{2}[0-9]{3}$')
);

-- Table d'association INSCRIPTIONS (relation N:M)
CREATE TABLE inscriptions (
    etudiant_id INTEGER NOT NULL,
    cours_id INTEGER NOT NULL,
    note NUMERIC(4,2),
    date_inscription DATE DEFAULT CURRENT_DATE,
    -- Clé primaire composée
    PRIMARY KEY (etudiant_id, cours_id),
    -- Clés étrangères
    FOREIGN KEY (etudiant_id)
        REFERENCES etudiants(id)
        ON DELETE CASCADE,
    FOREIGN KEY (cours_id)
        REFERENCES cours(id)
        ON DELETE CASCADE,
    -- Contrainte métier
    CONSTRAINT note_valide CHECK (note ⩾ 0 AND note ⩽ 20)
);
```

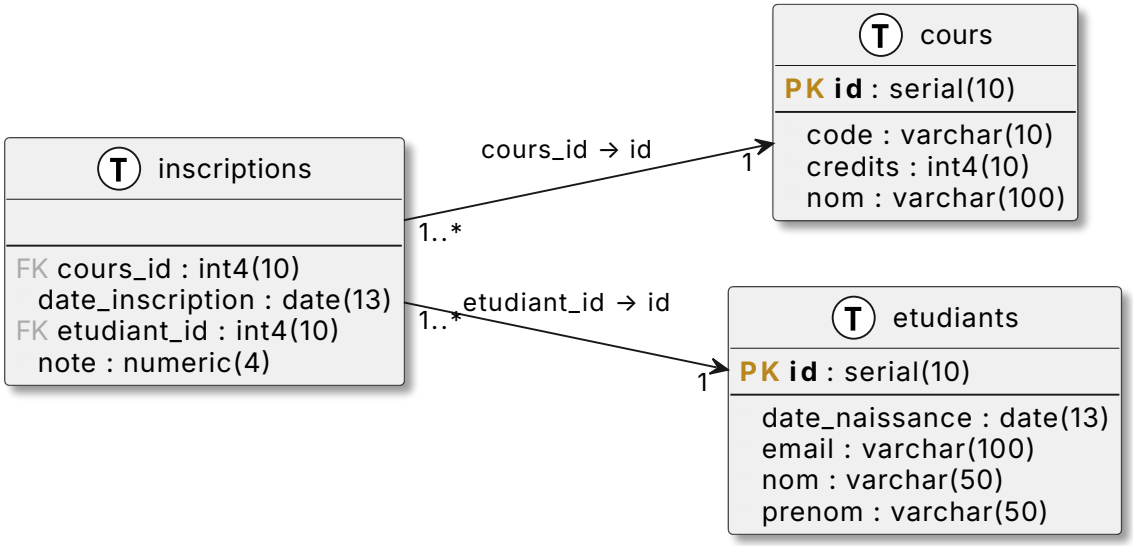
Updated 0 rows

Updated 0 rows

Updated 0 rows

Statements executed

```
%%rdbmsSchema
```




### 1.2.b Analyse détaillée du code

La gestion de la valeur des clés primaires, étrangères et des contraintes est cruciale pour garantir l'intégrité des données. Les clés primaires peuvent être gérées manuellement ou automatiquement. Dans tous les cas, le système vérifie leur unicité et leur non-nullité. La gestion automatique peut être réalisée de façon différentes parfois spécifiques à chaque SGBDR.

Les grandes solutions sont des mécanismes d'auto-incrémentation ou de séquences. Une séquence est un objet de base de données qui génère une suite de nombres uniques :

- PostgreSQL : type `SERIAL` ou `IDENTITY`
- MySQL : `AUTO_INCREMENT`

**Points techniques à noter** :

1. **SERIAL** :
   - Type PostgreSQL (non standard SQL)
   - Équivalent à `INTEGER NOT NULL DEFAULT nextval('sequence_name')`
   - Crée automatiquement une séquence auto-incrémentée
2. **Clé primaire composée** :

```
PRIMARY KEY (etudiant_id, cours_id)
```

   - Garantit l'unicité du couple
   - Un étudiant ne peut s'inscrire qu'une fois au même cours
3. **ON DELETE CASCADE** :

- Si un étudiant est supprimé → toutes ses inscriptions le sont aussi
- Alternative : `ON DELETE SET NULL`, `ON DELETE RESTRICT`

4. **Contraintes CHECK** :

```
CHECK (email LIKE '%@%.%')
CHECK (code ~ '^[A-Z]{2}[0-9]{3}$')
```

- Validation des données à l'insertion
- Expressions régulières POSIX avec `~`

Intégrité référentielle : Les clés étrangères garantissent que les références entre tables sont valides. Par conséquent, on ne peut pas insérer une inscription pour un étudiant ou un cours inexistant. On ne peut pas non plus supprimer un étudiant ou un cours référencé sans gérer les dépendances. Soit en supprimant manuellement les inscriptions associées, soit en utilisant `ON DELETE CASCADE` qui va le faire automatiquement. Attention cependant à l'usage de cette option qui peut entraîner des suppressions en cascade non désirées.

⚠️ ⚠ Ordre de création et suppression

**Toujours** respecter les dépendances :

- **Création** : tables référencées d'abord (étudiants, cours) puis tables référençantes (inscriptions)
- **Suppression** : ordre inverse (CASCADE permet d'ignorer cet ordre)

💡 Surrogate key

En pratique, on utilise souvent des clés primaires artificielles (surrogate keys) comme `id SERIAL` pour simplifier la gestion des relations, même si des clés naturelles existent (ex: email). Après la phase de modélisation, on choisit souvent une clé artificielle pour des raisons de performance et de simplicité. **Mais il faut toujours s'assurer que les contraintes d'unicité sur les clés naturelles sont bien en place.**

## 1.3 Insertion de données de test

Une fois le schéma créé, on peut insérer des données de test pour manipuler et interroger la base.

```
%%sqlAsTable
```

```sql
-- Insertion d'étudiants
INSERT INTO etudiants (nom, prenom, email, date_naissance) VALUES
    ('Dupont', 'Alice', 'alice.dupont@univ.fr', '2001-03-15'),
    ('Martin', 'Bob', 'bob.martin@univ.fr', '2000-11-22'),
    ('Bernard', 'Chloé', 'chloe.bernard@univ.fr', '2001-07-08'),
    ('Durand', 'David', 'david.durand@univ.fr', '2002-01-30'),
    ('Moreau', 'Emma', 'emma.moreau@univ.fr', '2001-05-19');

-- Insertion de cours
INSERT INTO cours (code, nom, credits) VALUES
    ('IF101', 'Bases de données', 6),
    ('IF102', 'Systèmes d''exploitation', 6),
    ('IF103', 'Programmation avancée', 6),
    ('IF104', 'Algorithmes et complexité', 5),
    ('IF105', 'Réseaux informatiques', 5),
    ('IF106', 'Architecture des ordinateurs', 4);

-- Inscriptions avec notes
INSERT INTO inscriptions (etudiant_id, cours_id, note) VALUES
    (1, 1, 15.5),
    (1, 2, 14.0),
    (1, 3, 16.5),
    (2, 1, 12.0),
    (2, 3, 13.5),
    (3, 1, 17.0),
    (3, 3, 16.0),
    (3, 4, 18.5),
    (4, 2, 11.0),
    (4, 4, 14.5),
    (5, 1, 15.0),
    (5, 2, 16.0),
    (5, 5, 17.5);
```

Updated 5 rows

Updated 6 rows

Updated 13 rows

Statements executed

### 1.3.a Visualisation des données insérées

**Les étudiants** :

```
%%sqlAsTable
SELECT * FROM etudiants;
```

| id | nom | prenom | email | date_naissance |
|---|---|---|---|---|
| 1 | Dupont | Alice | alice.dupont@univ.fr | 2001-03-15 |
| 2 | Martin | Bob | bob.martin@univ.fr | 2000-11-22 |
| 3 | Bernard | Chloé | chloe.bernard@univ.fr | 2001-07-08 |
| 4 | Durand | David | david.durand@univ.fr | 2002-01-30 |
| 5 | Moreau | Emma | emma.moreau@univ.fr | 2001-05-19 |

**Les cours** :

```
%%sqlAsTable
SELECT * FROM cours;
```

| id | code | nom | credits |
|---|---|---|---|
| 1 | IF101 | Bases de données | 6 |
| 2 | IF102 | Systèmes d'exploitation | 6 |
| 3 | IF103 | Programmation avancée | 6 |
| 4 | IF104 | Algorithmes et complexité | 5 |
| 5 | IF105 | Réseaux informatiques | 5 |
| 6 | IF106 | Architecture des ordinateurs | 4 |

**Les inscriptions** :

```
%%sqlAsTable
SELECT * FROM inscriptions;
```

| etudiant_id | cours_id | note | date_inscription |
|---|---|---|---|
| 1 | 1 | 15.50 | 2026-11-01 |
| 1 | 2 | 14.00 | 2026-11-01 |
| 1 | 3 | 16.50 | 2026-11-01 |
| 2 | 1 | 12.00 | 2026-11-01 |
| 2 | 3 | 13.50 | 2026-11-01 |

| etudiant_id | cours_id | note | date_inscription |
|---|---|---|---|
| 3 | 1 | 17.00 | 2026-11-01 |
| 3 | 3 | 16.00 | 2026-11-01 |
| 3 | 4 | 18.50 | 2026-11-01 |
| 4 | 2 | 11.00 | 2026-11-01 |
| 4 | 4 | 14.50 | 2026-11-01 |
| 5 | 1 | 15.00 | 2026-11-01 |
| 5 | 2 | 16.00 | 2026-11-01 |
| 5 | 5 | 17.50 | 2026-11-01 |

## 1.4 Exercices de compréhension du schéma

ℹ️ 🟡 Exercice 1.1 — Analyse des clés

**Questions** :

1. Identifiez toutes les clés primaires du schéma
2. Identifiez toutes les clés étrangères
3. Pourquoi la clé primaire d'`inscriptions` est-elle **composée** ?
4. Que se passe-t-il si on tente d'insérer `(1, 1, 18.0)` deux fois dans `inscriptions` ?

💡 Indice La clé primaire d'une table d'association reflète la **cardinalité** de la relation N:M. 🟡 Solution

1. **Clés primaires** :
   - `etudiants.id`
   - `cours.id`
   - `(inscriptions.etudiant_id, inscriptions.cours_id)`
2. **Clés étrangères** :
   - `inscriptions.etudiant_id` → `etudiants.id`
   - `inscriptions.cours_id` → `cours.id`
3. **Clé composée** : Car un étudiant peut s'inscrire à plusieurs cours ET un cours peut avoir plusieurs étudiants. La combinaison (étudiant, cours) est unique.
4. **Doublon** : La deuxième insertion échouera avec une erreur `duplicate key value violates unique constraint`.

ℹ️ Exercice 1.2 — Contraintes d'intégrité

Tester les requêtes suivantes et expliquer le résultat :

```sql
-- Test 1
INSERT INTO etudiants (nom, prenom, email)
VALUES ('Test', 'User', 'invalid-email');

-- Test 2
INSERT INTO cours (code, nom, credits)
VALUES ('invalid', 'Cours test', 5);

-- Test 3
INSERT INTO inscriptions VALUES (999, 1, 15);

-- Test 4
DELETE FROM etudiants WHERE id = 1;
```

🟡 Solutions **Test 1** : ✗ Erreur `CHECK constraint "email_valide" violated`

**Test 2** : ✗ Erreur `CHECK constraint "code_format" violated`

**Test 3** : ✗ Erreur `FOREIGN KEY constraint violated` (étudiant 999 n'existe pas)

**Test 4** : 🟡 Succès, mais **supprime aussi** toutes les inscriptions de l'étudiant 1 (CASCADE)

# 2 BLOC 2 : Requêtes SELECT — Fondamentaux et filtrage

La deuxième brique fondamentale du SQL est le langage de manipulation des données (DML - Data Manipulation Language). Le DML permet d'interroger et de manipuler les données stockées dans les tables via des requêtes.

## 2.1 Anatomie d'une requête SELECT

La commande `SELECT` est utilisée pour interroger les données dans une ou plusieurs tables.

Syntaxe formelle

```
SELECT [DISTINCT] liste_colonnes | *
FROM table(s)
[WHERE condition]
[GROUP BY colonnes]
[HAVING condition_agrégat]
[ORDER BY colonnes [ASC|DESC]]
[LIMIT n] [OFFSET m];
```

**Ordre d'exécution logique** (≠ ordre syntaxique) :

1. `FROM` : Sélection des tables source
2. `WHERE` : Filtrage des lignes
3. `GROUP BY` : Regroupement
4. `HAVING` : Filtrage des groupes
5. `SELECT` : Projection des colonnes
6. `ORDER BY` : Tri
7. `LIMIT/OFFSET` : Pagination

## 2.2 SELECT simple et projection

La version la plus simple de `SELECT` permet de récupérer des colonnes spécifiques ou toutes les colonnes d'une table. Elle correspond à l'opération de **projection** en algèbre relationnelle.

### 2.2.a Sélection de toutes les colonnes

```
%%sqlAsTable
SELECT * FROM etudiants;
```

| id | nom | prenom | email | date_naissance |
|---|---|---|---|---|
| 1 | Dupont | Alice | alice.dupont@univ.fr | 2001-03-15 |
| 2 | Martin | Bob | bob.martin@univ.fr | 2000-11-22 |
| 3 | Bernard | Chloé | chloe.bernard@univ.fr | 2001-07-08 |
| 4 | Durand | David | david.durand@univ.fr | 2002-01-30 |
| 5 | Moreau | Emma | emma.moreau@univ.fr | 2001-05-19 |

⚠️ ⚠️ Bonnes pratiques

**Évitez `SELECT *` en production** :

- **Performance** : Récupération de colonnes inutiles
- **Maintenance** : Modification du schéma → comportement imprévisible
- **Lisibilité** : On ne sait pas quelles colonnes sont utilisées

**Exception** : Exploration interactive des données (comme ici).

### 2.2.b Projection de colonnes spécifiques

```
%%sqlAsTable
SELECT nom, prenom, email
FROM etudiants;
```

| nom | prenom | email |
|---|---|---|
| Dupont | Alice | alice.dupont@univ.fr |
| Martin | Bob | bob.martin@univ.fr |
| Bernard | Chloé | chloe.bernard@univ.fr |
| Durand | David | david.durand@univ.fr |
| Moreau | Emma | emma.moreau@univ.fr |

### 2.2.c Alias de colonnes

```
%%sqlAsTable
SELECT
    nom AS nom_famille,
    prenom AS prenom_etudiant,
    EXTRACT(YEAR FROM AGE(date_naissance)) AS age
FROM etudiants;
```

| nom_famille | prenom_etudiant | age |
|---|---|---|
| Dupont | Alice | 25 |
| Martin | Bob | 25 |
| Bernard | Chloé | 25 |
| Durand | David | 24 |
| Moreau | Emma | 25 |

**Explications** :

- `AS` : Renomme une colonne dans le résultat
- `EXTRACT(YEAR FROM AGE(...))` : Fonction PostgreSQL pour calculer l'âge
- Utile pour clarifier les résultats ou pour des calculs

## 2.3 Clause WHERE : filtrage de lignes

La clause `WHERE` permet de filtrer les lignes selon des conditions logiques. Elle correspond à l'opération de **sélection** en algèbre relationnelle. L'opérateur `WHERE` peut utiliser des **opérateurs de comparaison** et des **opérateurs logiques** pour construire des conditions complexes. `LIKE` permet de faire du pattern matching avec des jokers (`%` pour plusieurs caractères, `_` pour un caractère).

### 2.3.a Opérateurs de comparaison

```
%%sqlAsTable
-- Étudiants dont le nom commence par 'D'
SELECT nom, prenom
FROM etudiants
WHERE nom LIKE 'D%';
```

| nom | prenom |
|---|---|
| Dupont | Alice |
| Durand | David |

**Opérateurs disponibles** :

| Opérateur | Description | Exemple |
|---|---|---|
| `=` | Égalité | `note = 15` |
| `<>` ou `≠` | Différence | `note <> 10` |
| `<`, `>`, `⩽`, `⩾` | Comparaison | `note ⩾ 12` |
| `BETWEEN` | Intervalle | `note BETWEEN 10 AND 15` |
| `IN` | Appartenance | `id IN (1, 2, 3)` |
| `LIKE` | Pattern matching | `nom LIKE 'D%'` |
| `~` | Regex POSIX | `email ~ '@univ\.fr$'` |
| `IS NULL` | Valeur nulle | `note IS NULL` |

### 2.3.b Opérateurs logiques

```
%%sqlAsTable
-- Étudiants nés après 2001 ET dont le prénom contient 'a'
SELECT nom, prenom, date_naissance
FROM etudiants
WHERE date_naissance > '2001-01-01'
  AND prenom ILIKE '%a%';  -- ILIKE : insensible à la casse
```

| nom | prenom | date_naissance |
|---|---|---|
| Dupont | Alice | 2001-03-15 |
| Durand | David | 2002-01-30 |
| Moreau | Emma | 2001-05-19 |

> 💡 Ordre de priorité
>
> Sans parenthèses : `NOT` > `AND` > `OR`
>
> ```
> -- Ces deux requêtes sont identiques
> WHERE condition1 AND condition2 OR condition3
> WHERE (condition1 AND condition2) OR condition3
>
> -- Utilisez des parenthèses pour la clarté !
> WHERE condition1 AND (condition2 OR condition3)
> ```

## 2.4 ORDER BY : tri des résultats

Le standard SQL ne garantit pas l'ordre des lignes retournées par une requête, pour obtenir un ordre spécifique, il faut utiliser la clause `ORDER BY`.

```
%%sqlAsTable
-- Étudiants triés par nom puis prénom
SELECT nom, prenom
FROM etudiants
ORDER BY nom ASC, prenom ASC;
```

| nom | prenom |
|---|---|
| Bernard | Chloé |
| Dupont | Alice |
| Durand | David |

| nom | prenom |
|---|---|
| Martin | Bob |
| Moreau | Emma |

```
%%sqlAsTable
-- Notes décroissantes avec information de cours
SELECT e.nom, e.prenom, c.nom AS cours, i.note
FROM inscriptions i
JOIN etudiants e ON e.id = i.etudiant_id
JOIN cours c ON c.id = i.cours_id
ORDER BY i.note DESC;
```

| nom | prenom | cours | note |
|---|---|---|---|
| Bernard | Chloé | Algorithmes et complexité | 18.50 |
| Moreau | Emma | Réseaux informatiques | 17.50 |
| Bernard | Chloé | Bases de données | 17.00 |
| Dupont | Alice | Programmation avancée | 16.50 |
| Bernard | Chloé | Programmation avancée | 16.00 |
| Moreau | Emma | Systèmes d'exploitation | 16.00 |
| Dupont | Alice | Bases de données | 15.50 |
| Moreau | Emma | Bases de données | 15.00 |
| Durand | David | Algorithmes et complexité | 14.50 |
| Dupont | Alice | Systèmes d'exploitation | 14.00 |
| Martin | Bob | Programmation avancée | 13.50 |
| Martin | Bob | Bases de données | 12.00 |
| Durand | David | Systèmes d'exploitation | 11.00 |

**Points importants** :

- `ASC` (ascendant) est la valeur par défaut
- On peut trier sur des colonnes **non sélectionnées**
- Les `NULL` sont triés en dernier par défaut en PostgreSQL
- Option `NULLS FIRST` ou `NULLS LAST` pour contrôler

## 2.5 LIMIT et OFFSET : pagination

Attention, par défaut, une requête `SELECT` retourne **toutes** les lignes correspondantes. Il est donc **important** de limiter le nombre de résultats pour éviter de surcharger le client ou le réseau. La clause `LIMIT` permet de restreindre le nombre de lignes retournées, tandis que `OFFSET` permet de sauter un certain nombre de lignes (utile pour la pagination). Il est **essentiel** d'utiliser `ORDER BY` avec `LIMIT` pour garantir un ordre cohérent des résultats.

```
%%sqlAsTable
-- Les 3 meilleures notes
SELECT e.nom, e.prenom, i.note
FROM inscriptions i
JOIN etudiants e ON e.id = i.etudiant_id
ORDER BY i.note DESC
LIMIT 3;
```

| nom | prenom | note |
|---|---|---|
| Bernard | Chloé | 18.50 |
| Moreau | Emma | 17.50 |
| Bernard | Chloé | 17.00 |

```
%%sqlAsTable
-- Pagination : page 2, taille 3
SELECT nom, prenom
FROM etudiants
ORDER BY nom
LIMIT 3 OFFSET 3;
```

| nom | prenom |
|---|---|
| Martin | Bob |
| Moreau | Emma |

**Formule pagination**

Pour une page $p$ de taille $n$ :

```
LIMIT n OFFSET (p-1)*n
```

Exemple : Page 3, taille 5 → `LIMIT 5 OFFSET 10`

## 2.6 Exercices — SELECT et filtrage

### ℹ️ Exercice 2.1 — Requêtes de base

Écrire les requêtes SQL pour :

1. Afficher les cours ayant 6 crédits
2. Afficher les étudiants avec un email Gmail (domaine différent)
3. Afficher les 5 étudiants les plus jeunes (triés par date de naissance décroissante)

▶ Solutions

```sql
-- 1. Cours avec 6 crédits
SELECT code, nom, credits
FROM cours
WHERE credits = 6;

-- 2. Étudiants Gmail (si applicable, adapter le domaine)
SELECT nom, prenom, email
FROM etudiants
WHERE email LIKE '%@gmail.com';

-- 3. 5 plus jeunes (date de naissance la plus récente)
SELECT nom, prenom, date_naissance
FROM etudiants
ORDER BY date_naissance DESC
LIMIT 5;
```

ℹ️ Exercice 2.2 — Filtrage avancé

1. Afficher les inscriptions avec une note entre 12 et 16 (inclus)
2. Afficher les étudiants dont le nom contient “ar” (insensible à la casse)
3. Afficher les cours dont le code commence par “IF10”

Solutions

```sql
-- 1. Notes entre 12 et 16
SELECT *
FROM inscriptions
WHERE note BETWEEN 12 AND 16;
-- Équivalent à : WHERE note ⩾ 12 AND note ⩽ 16

-- 2. Nom contenant 'ar' (insensible à la casse)
SELECT nom, prenom
FROM etudiants
WHERE nom ILIKE '%ar%';

-- 3. Codes commençant par 'IF10'
SELECT code, nom
FROM cours
WHERE code LIKE 'IF10%';
```

# 3 BLOC 3 : Jointures — Reconstruction de l’information

Pour interroger des données réparties sur plusieurs tables, on utilise des **jointures**. Les jointures permettent de combiner les lignes de deux ou plusieurs tables en fonction d’une condition logique.

## 3.1 Comprendre le problème : produit cartésien

Le produit cartésien de deux relations $R_1$ et $R_2$ est l’ensemble de toutes les combinaisons possibles de leurs tuples. Les jointures permettent de restreindre ce produit cartésien aux combinaisons pertinentes dans le contexte métier.

⚠️ ⚠️ Erreur classique : jointure oubliée

```
-- ✗ INCORRECT : Produit cartésien
SELECT e.nom, i.note
FROM etudiants e, inscriptions i;
```

**Résultat** : $|\text{etudiants}| \times |\text{inscriptions}| = 5 \times 13 = 65$ lignes !

**Problème** : Chaque étudiant est associé à **toutes** les notes, y compris celles des autres.

Testons pour observer l'erreur :

**ICI nous limitons le résultat à 10 lignes pour la lisibilité.** Pour voir l'erreur complète, il faut enlever le `LIMIT`.

```
%%sqlAsTable
-- Produit cartésien : 65 lignes au lieu de 13
SELECT e.nom, e.prenom, i.note
FROM etudiants e, inscriptions i
LIMIT 10;
```

| nom | prenom | note |
|---|---|---|
| Dupont | Alice | 15.50 |
| Martin | Bob | 15.50 |
| Bernard | Chloé | 15.50 |
| Durand | David | 15.50 |
| Moreau | Emma | 15.50 |
| Dupont | Alice | 14.00 |
| Martin | Bob | 14.00 |
| Bernard | Chloé | 14.00 |
| Durand | David | 14.00 |
| Moreau | Emma | 14.00 |

**Analyse** : Les notes ne correspondent pas aux bons étudiants ! Et les étudiants sont dupliqués.

## 3.2 INNER JOIN : jointure naturelle

`INNER JOIN` permet de combiner les lignes de deux tables en ne conservant que celles qui satisfont une condition de jointure (prédicat).

### 3.2.a Syntaxe moderne (recommandée)

```
%%sqlAsTable
-- Jointure correcte : chaque étudiant avec SES notes
SELECT e.nom, e.prenom, i.note
FROM etudiants e
INNER JOIN inscriptions i ON e.id = i.etudiant_id;
```

| nom | prenom | note |
|---|---|---|
| Dupont | Alice | 15.50 |
| Dupont | Alice | 14.00 |
| Dupont | Alice | 16.50 |
| Martin | Bob | 12.00 |
| Martin | Bob | 13.50 |
| Bernard | Chloé | 17.00 |
| Bernard | Chloé | 16.00 |
| Bernard | Chloé | 18.50 |
| Durand | David | 11.00 |
| Durand | David | 14.50 |
| Moreau | Emma | 15.00 |
| Moreau | Emma | 16.00 |
| Moreau | Emma | 17.50 |

**Explications** :

- `INNER JOIN` : Ne conserve que les lignes ayant une correspondance dans les deux tables
- `ON e.id = i.etudiant_id` : **Condition de jointure** (prédicat)
- Alias `e` et `i` : Évitent la répétition et améliorent la lisibilité

### 3.2.b Syntaxe ancienne (WHERE)

L'ancienne syntaxe utilise la clause `WHERE` pour définir la condition de jointure.

```
%%sqlAsTable
-- Équivalent avec WHERE (ancien style)
SELECT e.nom, e.prenom, i.note
FROM etudiants e, inscriptions i
WHERE e.id = i.etudiant_id;
```

| nom | prenom | note |
|---|---|---|
| Dupont | Alice | 15.50 |
| Dupont | Alice | 14.00 |
| Dupont | Alice | 16.50 |
| Martin | Bob | 12.00 |
| Martin | Bob | 13.50 |
| Bernard | Chloé | 17.00 |
| Bernard | Chloé | 16.00 |
| Bernard | Chloé | 18.50 |
| Durand | David | 11.00 |
| Durand | David | 14.50 |
| Moreau | Emma | 15.00 |
| Moreau | Emma | 16.00 |
| Moreau | Emma | 17.50 |

💡 Quelle syntaxe choisir ?

**Recommandation** : `INNER JOIN ... ON`

**Avantages** :

- Sépare clairement **jointure** (ON) et **filtrage** (WHERE)
- Meilleure lisibilité pour les requêtes complexes
- Norme SQL-92 (standard moderne)
- Préparation aux jointures externes (LEFT/RIGHT JOIN)

## 3.3 Jointures multiples

Il est courant de devoir joindre plusieurs tables pour reconstruire l'information complète.

```sql
%%sqlAsTable
-- Étudiant → Inscription → Cours (2 jointures)
SELECT
    e.nom,
    e.prenom,
    c.nom AS cours,
    i.note
FROM etudiants e
INNER JOIN inscriptions i ON e.id = i.etudiant_id
INNER JOIN cours c ON c.id = i.cours_id
ORDER BY e.nom, c.nom;
```

| nom | prenom | cours | note |
|---|---|---|---|
| Bernard | Chloé | Algorithmes et complexité | 18.50 |
| Bernard | Chloé | Bases de données | 17.00 |
| Bernard | Chloé | Programmation avancée | 16.00 |
| Dupont | Alice | Bases de données | 15.50 |
| Dupont | Alice | Programmation avancée | 16.50 |
| Dupont | Alice | Systèmes d'exploitation | 14.00 |
| Durand | David | Algorithmes et complexité | 14.50 |
| Durand | David | Systèmes d'exploitation | 11.00 |
| Martin | Bob | Bases de données | 12.00 |
| Martin | Bob | Programmation avancée | 13.50 |
| Moreau | Emma | Bases de données | 15.00 |
| Moreau | Emma | Réseaux informatiques | 17.50 |
| Moreau | Emma | Systèmes d'exploitation | 16.00 |

**Ordre d'exécution conceptuel** :

1. `FROM etudiants e` : Table de départ
2. `INNER JOIN inscriptions i` : Première jointure → résultat intermédiaire
3. `INNER JOIN cours c` : Deuxième jointure → résultat final

!🔍 Comprendre le flux de données

Visualisation :

```
etudiants (5 lignes)
    ↓ JOIN inscriptions (13 lignes)
    → Résultat intermédiaire (13 lignes : chaque inscription + son étudiant)
    ↓ JOIN cours (5 lignes)
    → Résultat final (13 lignes : chaque inscription + étudiant + cours)
```

## 3.4 LEFT JOIN : jointure externe gauche

### 3.4.a Problématique

Avec `INNER JOIN`, un cours sans inscription **disparaît** du résultat. La gestion de la nullité est cruciale dans les bases relationnelles.

```
%%sqlAsTable
-- Combien d'inscriptions par cours ?
SELECT c.nom, COUNT(i.etudiant_id) AS nb_inscrits
FROM cours c
INNER JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom;
```

| nom | nb_inscrits |
|---|---|
| Bases de données | 4 |
| Systèmes d'exploitation | 3 |
| Algorithmes et complexité | 2 |
| Programmation avancée | 3 |
| Réseaux informatiques | 1 |

**Problème** : Le cours « IF106 » (Architecture des ordinateurs) n'apparaît car il n'a aucune inscription.

### 3.4.b Solution : LEFT JOIN

```
%%sqlAsTable
-- Tous les cours, même sans inscription
SELECT
    c.nom AS cours,
    COUNT(i.etudiant_id) AS nb_inscrits
```

```
FROM cours c
LEFT JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom
ORDER BY nb_inscrits DESC;
```

| cours | nb_inscrits |
|---|---|
| Bases de données | 4 |
| Systèmes d'exploitation | 3 |
| Programmation avancée | 3 |
| Algorithmes et complexité | 2 |
| Réseaux informatiques | 1 |
| Architecture des ordinateurs | 0 |

**Définition : LEFT JOIN**

`A LEFT JOIN B ON condition` retourne :

- Toutes les lignes de `A`
- Les lignes correspondantes de `B` (si elles existent)
- `NULL` pour les colonnes de `B` si aucune correspondance

**Mnémotechnique** : « Tout à **gauche**, complété à droite »

## 3.5 Autres types de jointures

Il existe d'autres types de jointures pour des besoins spécifiques.

### 3.5.a RIGHT JOIN

La jointure externe droite est l'inverse de `LEFT JOIN`. Elle conserve toutes les lignes de la table de droite.

```
%%sqlAsTable
-- Symétrique de LEFT JOIN
SELECT c.nom, e.nom
FROM inscriptions i
RIGHT JOIN cours c ON c.id = i.cours_id
RIGHT JOIN etudiants e ON e.id = i.etudiant_id;
```

| nom | nom |
|---|---|
| Bases de données | Dupont |

| nom | nom |
|---|---|
| Systèmes d'exploitation | Dupont |
| Programmation avancée | Dupont |
| Bases de données | Martin |
| Programmation avancée | Martin |
| Bases de données | Bernard |
| Programmation avancée | Bernard |
| Algorithmes et complexité | Bernard |
| Systèmes d'exploitation | Durand |
| Algorithmes et complexité | Durand |
| Bases de données | Moreau |
| Systèmes d'exploitation | Moreau |
| Réseaux informatiques | Moreau |

> 💡 Bonnes pratiques
>
> `RIGHT JOIN` est **rarement utilisé** en pratique :
>
> - Moins intuitif (lecture de gauche à droite)
> - `LEFT JOIN` équivalent en inversant l'ordre des tables
> - **Recommandation** : Toujours utiliser `LEFT JOIN` pour la cohérence

### 3.5.b FULL OUTER JOIN

La jointure externe complète combine `LEFT JOIN` et `RIGHT JOIN`. Elle conserve toutes les lignes des deux tables, avec `NULL` pour les colonnes sans correspondance.

```
%%sqlAsTable
-- Tous les étudiants ET tous les cours (avec ou sans lien)
SELECT
    e.nom AS etudiant,
    c.nom AS cours,
    i.note
FROM etudiants e
FULL OUTER JOIN inscriptions i ON e.id = i.etudiant_id
FULL OUTER JOIN cours c ON c.id = i.cours_id;
```

| etudiant | cours | note |
|---|---|---|
| Dupont | Bases de données | 15.50 |
| Dupont | Systèmes d'exploitation | 14.00 |
| Dupont | Programmation avancée | 16.50 |
| Martin | Bases de données | 12.00 |
| Martin | Programmation avancée | 13.50 |
| Bernard | Bases de données | 17.00 |
| Bernard | Programmation avancée | 16.00 |
| Bernard | Algorithmes et complexité | 18.50 |
| Durand | Systèmes d'exploitation | 11.00 |
| Durand | Algorithmes et complexité | 14.50 |
| Moreau | Bases de données | 15.00 |
| Moreau | Systèmes d'exploitation | 16.00 |
| Moreau | Réseaux informatiques | 17.50 |
| | Architecture des ordinateurs | |

**Utilité** : Rare, utile pour détecter les données orphelines.

## 3.6 Exercices — Jointures

ℹ️ Exercice 3.1 — Jointures simples

1. Afficher le nom complet des étudiants avec le nom de chaque cours suivi
2. Afficher les étudiants inscrits au cours « Bases de données »
3. Afficher les cours suivis par « Alice Dupont »

▌Solutions

```sql
-- 1. Étudiants et leurs cours
SELECT
    e.nom || ' ' || e.prenom AS nom_complet,
    c.nom AS cours
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
JOIN cours c ON c.id = i.cours_id
ORDER BY nom_complet, cours;

-- 2. Étudiants en BD
SELECT DISTINCT e.nom, e.prenom
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
JOIN cours c ON c.id = i.cours_id
WHERE c.nom = 'Bases de données';

-- 3. Cours d'Alice Dupont
SELECT c.nom, c.code
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
JOIN etudiants e ON e.id = i.etudiant_id
WHERE e.nom = 'Dupont' AND e.prenom = 'Alice';
```

ℹ️ Exercice 3.2 — Jointures externes

1. Lister tous les étudiants avec leur nombre d'inscriptions (0 si aucune)
2. Trouver les cours sans aucune inscription
3. Trouver les étudiants non inscrits à aucun cours

Solutions

```
-- 1. Étudiants et nombre d'inscriptions
SELECT
    e.nom,
    e.prenom,
    COUNT(i.cours_id) AS nb_cours
FROM etudiants e
LEFT JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY e.id, e.nom, e.prenom
ORDER BY nb_cours DESC;

-- 2. Cours sans inscription
SELECT c.code, c.nom
FROM cours c
LEFT JOIN inscriptions i ON c.id = i.cours_id
WHERE i.cours_id IS NULL;

-- 3. Étudiants sans inscription
SELECT e.nom, e.prenom
FROM etudiants e
LEFT JOIN inscriptions i ON e.id = i.etudiant_id
WHERE i.etudiant_id IS NULL;
```

# 4 BLOC 4 : Fonctions d'agrégat et GROUP BY

## 4.1 Fonctions d'agrégat : résumer les données

SQL propose des **fonctions d'agrégat** pour calculer des valeurs résumées à partir d'un ensemble de lignes (ex: moyenne, somme, minimum, maximum, etc.).

### 4.1.a Fonctions de base

PostgreSQL fournit des **fonctions d'agrégat** qui calculent une valeur unique à partir d'un ensemble de lignes.

```
%%sqlAsTable
-- Statistiques globales sur les notes
```

```sql
SELECT
    COUNT(*) AS nb_inscriptions,
    COUNT(note) AS nb_notes,  -- Ignore les NULL
    AVG(note) AS moyenne_generale,
    MIN(note) AS note_min,
    MAX(note) AS note_max,
    STDDEV(note) AS ecart_type
FROM inscriptions;
```

| nb_inscriptions | nb_notes | moyenne_generale | note_min | note_max | ecart_type |
|---|---|---|---|---|---|
| 13 | 13 | 15.153846153846153 | 11.00 | 18.50 | 2.1446116138730048 |

**Fonctions disponibles** :

| Fonction | Description | NULL |
|---|---|---|
| `COUNT(*)` | Nombre de lignes | Compte tout |
| `COUNT(col)` | Valeurs non NULL | Ignore NULL |
| `SUM(col)` | Somme | Ignore NULL |
| `AVG(col)` | Moyenne | Ignore NULL |
| `MIN(col)` | Minimum | Ignore NULL |
| `MAX(col)` | Maximum | Ignore NULL |
| `STDDEV(col)` | Écart-type | Ignore NULL |
| `VARIANCE(col)` | Variance | Ignore NULL |

⚠️ ⚠ Différence COUNT(*) vs COUNT(colonne)

```
-- Si une ligne a note = NULL :
COUNT(*)      → 1 (compte la ligne)
COUNT(note)   → 0 (ignore NULL)
```

## 4.2 GROUP BY : agrégation par groupe

Les tables contiennent souvent des données réparties sur plusieurs lignes. Pour obtenir des statistiques **par groupe** (ex: par étudiant, par cours), on utilise la clause `GROUP BY`. Par exemple, les notes d'un étudiant sont réparties sur plusieurs lignes (une par cours). Pour calculer la moyenne par étudiant, on doit regrouper les lignes par étudiant.

### 4.2.a Principe

`GROUP BY` divise les lignes en **groupes** et applique les agrégats à chaque groupe.

```
%%sqlAsTable
-- Moyenne par étudiant
SELECT
    e.nom,
    e.prenom,
    COUNT(i.cours_id) AS nb_cours,
    AVG(i.note) AS moyenne,
    MIN(i.note) AS note_min,
    MAX(i.note) AS note_max
FROM etudiants e
LEFT JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY e.id, e.nom, e.prenom
ORDER BY moyenne DESC NULLS LAST;
```

| nom | prenom | nb_cours | moyenne | note_min | note_max |
|---|---|---|---|---|---|
| Bernard | Chloé | 3 | 17.1666666666666667 | 16.00 | 18.50 |
| Moreau | Emma | 3 | 16.1666666666666667 | 15.00 | 17.50 |
| Dupont | Alice | 3 | 15.3333333333333333 | 14.00 | 16.50 |
| Durand | David | 2 | 12.7500000000000000 | 11.00 | 14.50 |
| Martin | Bob | 2 | 12.7500000000000000 | 12.00 | 13.50 |

!🔍 Règle GROUP BY en PostgreSQL

**Toute colonne du SELECT non agrégée doit figurer dans GROUP BY.**

```
-- ✕ INCORRECT
SELECT nom, prenom, AVG(note)
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY nom;  -- Manque prenom !

-- ☑ CORRECT
SELECT nom, prenom, AVG(note)
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY nom, prenom;  -- Ou mieux : GROUP BY e.id, nom, prenom
```

**Pourquoi ?** Évite l'ambiguïté : si deux personnes ont le même nom mais des prénoms différents, quel prénom afficher ?

### 4.2.b Groupement sur clé primaire

Le groupement peut être simplifié si on groupe sur une **clé primaire**. En effet, une clé primaire détermine de façon unique toutes les autres colonnes de la ligne.

```
%%sqlAsTable
-- Grouper sur l'ID suffit (ID détermine nom et prenom)
SELECT
    e.id,
    e.nom,
    e.prenom,
    COUNT(i.cours_id) AS nb_cours
FROM etudiants e
LEFT JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY e.id;  -- Suffit car id → nom, prenom (dépendance fonctionnelle)
```

| id | nom | prenom | nb_cours |
|---|---|---|---|
| 3 | Bernard | Chloé | 3 |
| 5 | Moreau | Emma | 3 |
| 4 | Durand | David | 2 |
| 2 | Martin | Bob | 2 |
| 1 | Dupont | Alice | 3 |

**Dépendance fonctionnelle**

Si `id` est clé primaire, alors `id → nom, prenom`.

PostgreSQL accepte :

```
GROUP BY e.id  -- Au lieu de GROUP BY e.id, e.nom, e.prenom
```

Mais **préférez lister toutes les colonnes** pour la portabilité.

## 4.3 HAVING : filtrage après agrégation

### 4.3.a Problématique

`WHERE` filtre **avant** l'agrégation. Pour filtrer **après**, on utilise `HAVING`.

```
-- ✗ INCORRECT : WHERE sur un agrégat
SELECT c.nom, AVG(i.note) AS moyenne
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
WHERE AVG(i.note) ⩾ 14  -- ERREUR !
GROUP BY c.nom;
```

**Erreur** : `aggregate functions are not allowed in WHERE`

### 4.3.b Solution : HAVING

```
%%sqlAsTable
-- ☑ CORRECT : HAVING filtre les groupes
SELECT
    c.nom AS cours,
    COUNT(i.etudiant_id) AS nb_inscrits,
    AVG(i.note) AS moyenne
FROM cours c
LEFT JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom
HAVING COUNT(i.etudiant_id) > 0  -- Cours avec au moins 1 inscription
   AND AVG(i.note) ⩾ 14          -- Moyenne ⩾ 14
ORDER BY moyenne DESC;
```

| cours | nb_inscrits | moyenne |
|---|---|---|
| Réseaux informatiques | 1 | 17.5000000000000000 |
| Algorithmes et complexité | 2 | 16.5000000000000000 |

| cours | nb_inscrits | moyenne |
|---|---|---|
| Programmation avancée | 3 | 15.3333333333333333 |
| Bases de données | 4 | 14.8750000000000000 |

> 📓 Ordre d'exécution complet
>
> ```
> 1. FROM + JOIN      → Construction des lignes sources
> 2. WHERE            → Filtrage des lignes
> 3. GROUP BY         → Formation des groupes
> 4. HAVING           → Filtrage des groupes
> 5. SELECT           → Calcul des colonnes (avec agrégats)
> 6. ORDER BY         → Tri
> 7. LIMIT / OFFSET   → Pagination
> ```

## 4.4 Exemples avancés

### 4.4.a Nombre d'étudiants par cours avec détails

```
%%sqlAsTable
SELECT
    c.code,
    c.nom,
    c.credits,
    COUNT(i.etudiant_id) AS nb_etudiants,
    ROUND(AVG(i.note), 2) AS moyenne,
    STRING_AGG(e.nom || ' ' || e.prenom, ', ' ORDER BY e.nom) AS liste_etudiants
FROM cours c
LEFT JOIN inscriptions i ON c.id = i.cours_id
LEFT JOIN etudiants e ON e.id = i.etudiant_id
GROUP BY c.id, c.code, c.nom, c.credits
ORDER BY nb_etudiants DESC;
```

| code | nom | credits | nb_etudiants | moyenne | liste_etudiants |
|---|---|---|---|---|---|
| IF101 | Bases de données | 6 | 4 | 14.88 | Bernard Chloé, Dupont Alice, Martin Bob, Moreau Emma |

| code | nom | credits | nb_etudiants | moyenne | liste_etudiants |
|---|---|---|---|---|---|
| IF102 | Systèmes d'exploitation | 6 | 3 | 13.67 | Dupont Alice, Durand David, Moreau Emma |
| IF103 | Programmation avancée | 6 | 3 | 15.33 | Bernard Chloé, Dupont Alice, Martin Bob |
| IF104 | Algorithmes et complexité | 5 | 2 | 16.50 | Bernard Chloé, Durand David |
| IF105 | Réseaux informatiques | 5 | 1 | 17.50 | Moreau Emma |
| IF106 | Architecture des ordinateurs | 4 | 0 | | |

**Nouvelles fonctions** :

- `ROUND(valeur, decimales)` : Arrondi
- `STRING_AGG(colonne, separateur ORDER BY ...)` : Concaténation agrégée

## 4.5 Exercices — Agrégats

### ℹ️ Exercice 4.1 — Statistiques simples

1. Calculer le nombre total de crédits proposés (somme sur tous les cours)
2. Trouver la note minimale et maximale de chaque cours
3. Compter le nombre d'étudiants ayant une moyenne >= 15

▌Solutions

```sql
-- 1. Total des crédits
SELECT SUM(credits) AS total_credits
FROM cours;

-- 2. Min/max par cours
SELECT
    c.nom,
    MIN(i.note) AS note_min,
    MAX(i.note) AS note_max
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom;

-- 3. Étudiants avec moyenne ⩾ 15
SELECT COUNT(*) AS nb_etudiants
FROM (
    SELECT e.id
    FROM etudiants e
    JOIN inscriptions i ON e.id = i.etudiant_id
    GROUP BY e.id
    HAVING AVG(i.note) ⩾ 15
) AS subquery;
```

ℹ️ Exercice 4.2 — Analyses complexes

1. Afficher les cours dont la moyenne est supérieure à la moyenne générale
2. Trouver l'étudiant avec le plus de cours suivis
3. Calculer le taux de réussite (note >= 10) par cours

▶ Solutions

```sql
-- 1. Cours au-dessus de la moyenne générale
SELECT c.nom, AVG(i.note) AS moyenne_cours
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom
HAVING AVG(i.note) > (SELECT AVG(note) FROM inscriptions);

-- 2. Étudiant avec le plus de cours
SELECT e.nom, e.prenom, COUNT(i.cours_id) AS nb_cours
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY e.id, e.nom, e.prenom
ORDER BY nb_cours DESC
LIMIT 1;

-- 3. Taux de réussite par cours
SELECT
    c.nom,
    COUNT(*) AS nb_etudiants,
    COUNT(*) FILTER (WHERE i.note >= 10) AS nb_reussis,
      ROUND(100.0 * COUNT(*) FILTER (WHERE i.note >= 10) / COUNT(*), 2)
AS taux_reussite
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
GROUP BY c.nom;
```

**Note** : `FILTER (WHERE ...)` est une syntaxe PostgreSQL pour les agrégats conditionnels.

# 5 BLOC 5 : Sous-requêtes et requêtes avancées

## 5.1 Sous-requêtes scalaires

Une **sous-requête scalaire** retourne une seule valeur (1 ligne, 1 colonne).

```
%%sqlAsTable
-- Étudiants ayant une note supérieure à la moyenne générale
SELECT e.nom, e.prenom, i.note
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
WHERE i.note > (SELECT AVG(note) FROM inscriptions)
ORDER BY i.note DESC;
```

| nom | prenom | note |
|---|---|---|
| Bernard | Chloé | 18.50 |
| Moreau | Emma | 17.50 |
| Bernard | Chloé | 17.00 |
| Dupont | Alice | 16.50 |
| Bernard | Chloé | 16.00 |
| Moreau | Emma | 16.00 |
| Dupont | Alice | 15.50 |

## 5.2 Sous-requêtes avec IN

La clause `IN` permet de vérifier l'appartenance à un ensemble retourné par une sous-requête.

```
%%sqlAsTable
-- Étudiants inscrits à au moins un cours de 6 crédits
SELECT DISTINCT e.nom, e.prenom
FROM etudiants e
WHERE e.id IN (
    SELECT i.etudiant_id
    FROM inscriptions i
    JOIN cours c ON c.id = i.cours_id
    WHERE c.credits = 6
);
```

| nom | prenom |
|---|---|
| Bernard | Chloé |
| Dupont | Alice |
| Durand | David |
| Martin | Bob |

| nom | prenom |
|---|---|
| Moreau | Emma |

**Alternative avec JOIN** (souvent plus performante) :

```
%%sqlAsTable
SELECT DISTINCT e.nom, e.prenom
FROM etudiants e
JOIN inscriptions i ON e.id = i.etudiant_id
JOIN cours c ON c.id = i.cours_id
WHERE c.credits = 6;
```

| nom | prenom |
|---|---|
| Bernard | Chloé |
| Dupont | Alice |
| Durand | David |
| Martin | Bob |
| Moreau | Emma |

## 5.3 Sous-requêtes corrélées

La clause `EXISTS` teste l'existence de lignes retournées par une sous-requête.

Une sous-requête **corrélée** référence la requête externe.

```
%%sqlAsTable
-- Étudiants ayant au moins une note ⩾ 15
SELECT e.nom, e.prenom
FROM etudiants e
WHERE EXISTS (
    SELECT 1
    FROM inscriptions i
    WHERE i.etudiant_id = e.id
      AND i.note ⩾ 15
);
```

| nom | prenom |
|---|---|
| Dupont | Alice |
| Bernard | Chloé |
| Moreau | Emma |

⚠️ ⚠️ Performance

Les sous-requêtes corrélées peuvent être **coûteuses** (exécutées N fois).

**Astuce** : Vérifiez avec `EXPLAIN ANALYZE` (voir BLOC 6).

## 5.4 CTE (Common Table Expressions)

Les **CTE** (clause `WITH`) améliorent la lisibilité.

```
%%sqlAsTable
-- Calcul de la moyenne par étudiant, puis sélection des meilleurs
WITH moyennes AS (
    SELECT
        e.id,
        e.nom,
        e.prenom,
        AVG(i.note) AS moyenne
    FROM etudiants e
    JOIN inscriptions i ON e.id = i.etudiant_id
    GROUP BY e.id, e.nom, e.prenom
)
SELECT nom, prenom, ROUND(moyenne, 2) AS moyenne
FROM moyennes
WHERE moyenne ⩾ 15
ORDER BY moyenne DESC;
```

| nom | prenom | moyenne |
|---|---|---|
| Bernard | Chloé | 17.17 |
| Moreau | Emma | 16.17 |
| Dupont | Alice | 15.33 |

**Avantages** :

- Code modulaire et réutilisable
- Équivalent à une sous-requête dans `FROM`, mais plus lisible
- Peut être **récursive** (CTE RECURSIVE)

## 5.5 UNION et opérations ensemblistes

SQL permet de combiner les résultats de plusieurs requêtes avec des opérations ensemblistes.

```
%%sqlAsTable
-- Tous les emails (étudiants + fictif pour cours)
SELECT email AS contact, 'Étudiant' AS type
FROM etudiants
UNION
SELECT code || '@cours.univ.fr', 'Cours'
FROM cours
ORDER BY type, contact;
```

| contact | type |
|---|---|
| IF101@cours.univ.fr | Cours |
| IF102@cours.univ.fr | Cours |
| IF103@cours.univ.fr | Cours |
| IF104@cours.univ.fr | Cours |
| IF105@cours.univ.fr | Cours |
| IF106@cours.univ.fr | Cours |
| alice.dupont@univ.fr | Étudiant |
| bob.martin@univ.fr | Étudiant |
| chloe.bernard@univ.fr | Étudiant |
| david.durand@univ.fr | Étudiant |
| emma.moreau@univ.fr | Étudiant |

**Opérateurs** :

- `UNION` : Union (dédoublonnée)
- `UNION ALL` : Union (avec doublons)
- `INTERSECT` : Intersection
- `EXCEPT` : Différence

# 6 BLOC 6 : Performance et optimisation

## 6.1 Index : accélérer les recherches

### 6.1.a Principe

Un **index** est une structure de données auxiliaire permettant des recherches rapides.

> 📖 Analogie : Index d'un livre
>
> - **Sans index** : Lire toutes les pages pour trouver un mot (parcours séquentiel)
> - **Avec index** : Consulter l'index alphabétique → aller directement à la page (accès direct)
>
> **Complexité** :
>
> - Sans index : $O(n)$ (scan complet)
> - Avec index : $O(\log n)$ (arbre B)

#### 6.1.b Création d'index

```
%%sqlAsTable
-- Index sur email (recherche fréquente)
CREATE INDEX idx_etudiants_email ON etudiants(email);

-- Index sur clé étrangère (jointures fréquentes)
CREATE INDEX idx_inscriptions_etudiant ON inscriptions(etudiant_id);
CREATE INDEX idx_inscriptions_cours ON inscriptions(cours_id);

-- Index composé
CREATE INDEX idx_inscriptions_etudiant_cours ON inscriptions(etudiant_id,
cours_id);
```

Statements executed

**Types d'index PostgreSQL** :

- **B-tree** (défaut) : Égalité, comparaison, ORDER BY
- **Hash** : Égalité uniquement
- **GIN** : Recherche plein texte, tableaux, JSON
- **GiST** : Données spatiales

### 6.2 EXPLAIN ANALYZE : comprendre l'exécution

Pour exécuter un moteur de base de données efficacement, il est crucial de comprendre comment une requête est exécutée. PostgreSQL fournit la commande `EXPLAIN` pour afficher le plan d'exécution d'une requête, et `EXPLAIN ANALYZE` pour exécuter la requête et fournir des statistiques réelles.

```
%%sqlAsTable
EXPLAIN ANALYZE
SELECT e.nom, e.prenom, AVG(i.note) AS moyenne
FROM etudiants e
```

```
JOIN inscriptions i ON e.id = i.etudiant_id
GROUP BY e.id, e.nom, e.prenom
HAVING AVG(i.note) ⩾ 15;
```

QUERY PLAN

HashAggregate (cost=2.35..2.43 rows=2 width=272) (actual time=0.047..0.051 rows=3 loops=1)

Group Key: e.id

Filter: (avg(i.note) >= '15'::numeric)

Batches: 1 Memory Usage: 24kB

Rows Removed by Filter: 2

→ Hash Join (cost=1.11..2.29 rows=13 width=252) (actual time=0.031..0.037 rows=13 loops=1)

Hash Cond: (i.etudiant_id = e.id)

→ Seq Scan on inscriptions i (cost=0.00..1.13 rows=13 width=16) (actual time=0.008..0.009 rows=13 loops=1)

→ Hash (cost=1.05..1.05 rows=5 width=240) (actual time=0.010..0.010 rows=5 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

→ Seq Scan on etudiants e (cost=0.00..1.05 rows=5 width=240) (actual time=0.003..0.004 rows=5 loops=1)

Planning Time: 0.140 ms

Execution Time: 0.084 ms

**Lecture du plan d'exécution** :

- **Seq Scan** : Parcours séquentiel (toutes les lignes)
- **Index Scan** : Utilisation d'un index
- **Nested Loop** : Jointure par boucle imbriquée
- **Hash Join** : Jointure par table de hachage
- **Cost** : Estimation du coût (arbitraire)
- **Actual time** : Temps réel (avec ANALYZE)

💡 Interpréter le coût

```
cost=0.00..35.50  →  Coût de démarrage..coût total
rows=1000         →  Nombre de lignes estimé
width=64          →  Taille moyenne d'une ligne (octets)
```

**Attention** : Le coût est **relatif**, pas un temps absolu.

## 6.3 Exercices — Optimisation

ℹ️ Exercice 6.1 — Analyse de performance

1. Exécuter `EXPLAIN` sur la requête suivante :

```sql
SELECT * FROM inscriptions WHERE etudiant_id = 2;
```

2. Créer un index sur `etudiant_id`
3. Ré-exécuter `EXPLAIN` et comparer

Démarche

```sql
-- Sans index
EXPLAIN SELECT * FROM inscriptions WHERE etudiant_id = 2;
-- → Seq Scan (parcours complet)

-- Création de l'index
CREATE INDEX idx_inscriptions_etudiant ON inscriptions(etudiant_id);

-- Avec index
EXPLAIN SELECT * FROM inscriptions WHERE etudiant_id = 2;
-- → Index Scan (accès direct)
```

# 7 BLOC 7 : Conception et normalisation

## 7.1 Anomalies de mise à jour

### 7.1.a Schéma dénormalisé (mauvais)

```sql
-- ✗ Table unique : redondance massive
CREATE TABLE inscriptions_denorm (
    etudiant_nom VARCHAR,
```

```
    etudiant_prenom VARCHAR,
    etudiant_email VARCHAR,
    cours_code VARCHAR,
    cours_nom VARCHAR,
    cours_credits INTEGER,
    note NUMERIC
);
```

**Problèmes** :

1. **Anomalie d'insertion** : Impossible d'ajouter un cours sans étudiant
2. **Anomalie de modification** : Changer le nom d'un cours → modifier N lignes
3. **Anomalie de suppression** : Supprimer le dernier étudiant → perte du cours
4. **Redondance** : Si 100 étudiants suivent un cours, le nom du cours est dupliqué 100 fois

## 7.2 Formes normales

### 1NF Première Forme Normale (1NF)

**Définition** : Tous les attributs sont **atomiques** (indivisibles).

**✗ Violation** :

```
CREATE TABLE etudiants_mauvais (
    id INTEGER PRIMARY KEY,
    nom VARCHAR,
    cours_suivis VARCHAR  -- "BD,SE,Algo" → NON ATOMIQUE
);
```

**✓ Conforme 1NF** : Table d'association `inscriptions`

### 🟦 Deuxième Forme Normale (2NF)

**Définition** : 1NF + aucun attribut non-clé ne dépend d'une **partie** de la clé.

**✗ Violation** :

```
CREATE TABLE inscriptions_2nf (
    etudiant_id INTEGER,
    cours_id INTEGER,
    note NUMERIC,
    cours_nom VARCHAR,  -- Dépend uniquement de cours_id !
    PRIMARY KEY (etudiant_id, cours_id)
);
```

🟦 **Conforme 2NF** : Séparer `cours_nom` dans la table `cours`

### 🟦 Troisième Forme Normale (3NF)

**Définition** : 2NF + aucune dépendance transitive.

**✗ Violation** :

```
CREATE TABLE etudiants_3nf (
    id INTEGER PRIMARY KEY,
    nom VARCHAR,
    ville VARCHAR,
    code_postal VARCHAR,  -- ville → code_postal (transitive)
);
```

🟦 **Conforme 3NF** : Créer une table `villes(nom, code_postal)`

## 7.3 Exercice de conception

ℹ️ Atelier final — Système de bibliothèque

**Cahier des charges** :

Une bibliothèque gère des **livres** et des **emprunts** par des **membres**.

- Un livre a un titre, un auteur, un ISBN (unique), une année de publication
- Un membre a un nom, prénom, email (unique), date d'inscription
- Un emprunt enregistre : qui, quel livre, date d'emprunt, date de retour prévue, date de retour réelle

**Travail demandé** :

1. Concevoir le schéma relationnel (3 tables minimum)
2. Identifier toutes les clés (primaires, étrangères)
3. Définir les contraintes (NOT NULL, CHECK, etc.)
4. Implémenter en SQL (CREATE TABLE)
5. Insérer 3 livres, 2 membres, 4 emprunts
6. Écrire 3 requêtes pertinentes :
   - Livres actuellement empruntés
   - Membres n'ayant jamais emprunté
   - Livres les plus empruntés

💡 Indice — Structure Tables suggérées :

- `livres(id, isbn, titre, auteur, annee)`
- `membres(id, nom, prenom, email, date_inscription)`
- `emprunts(id, livre_id, membre_id, date_emprunt, date_retour_prevue, date_retour_reelle)`

Réfléchissez aux cardinalités : un livre peut être emprunté plusieurs fois (mais pas en même temps), un membre peut emprunter plusieurs livres. ▓ Solution complète

```
-- Création du schéma
CREATE TABLE livres (
    id SERIAL PRIMARY KEY,
    isbn VARCHAR(17) UNIQUE NOT NULL,
    titre VARCHAR(200) NOT NULL,
    auteur VARCHAR(100) NOT NULL,
      annee INTEGER CHECK (annee > 1000 AND annee ⩽ EXTRACT(YEAR FROM
CURRENT_DATE))
);
```

```
CREATE TABLE membres (
    id SERIAL PRIMARY KEY,
    nom VARCHAR(50) NOT NULL,
    prenom VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    date_inscription DATE DEFAULT CURRENT_DATE
```

# 8 BLOC 8 : Transactions et isolation

## 8.1 Introduction aux transactions : Garantir la cohérence

Bien que les SGBD garantissent l'atomicité au niveau de la requête SQL individuelle (une instruction est exécutée soit totalement, soit pas du tout), cette protection est insuffisante pour les processus métier complexes. Une opération logique (ex: un virement bancaire) nécessite souvent plusieurs instructions interdépendantes : débiter un compte, puis en créditer un autre.

Si une interruption survient entre ces deux étapes, la base de données bascule dans un état incohérent. La transaction intervient alors comme une unité logique de travail qui regroupe ces opérations. Les défis de la fiabilité

La gestion des transactions doit répondre à trois problématiques majeures :

- L'atomicité multi-requêtes : Garantir que l'ensemble du bloc d'instructions est validé (Commit) ou annulé en cas d'erreur (Rollback), sans état intermédiaire.
- La concurrence d'accès : Dans un environnement multi-utilisateurs, l'exécution simultanée de plusieurs transactions peut provoquer des anomalies (lecture sale, lecture non reproductible). Le SGBD doit assurer l'Isolation de ces traitements.
- La résilience aux pannes : En cas de crash système (panne matérielle ou logicielle), le mécanisme de journalisation (Write-Ahead Logging) doit permettre de reconstruire un état stable, garantissant ainsi la Durabilité des données.

### 8.1.a Définition et propriétés ACID

> 📘 Définition : Transaction
>
> Une **transaction** est une séquence d'opérations SQL formant une **unité logique de travail** qui s'exécute de manière atomique.
>
> **Principe** : Tout ou rien (all or nothing)

Dans le cadre des bases de données relationnelles, les transactions garantissent les propriétés **ACID** :

| Propriété | Description | Exemple |
|---|---|---|
| **Atomicité** | Tout ou rien | Virement bancaire : débit ET crédit, pas l'un sans l'autre |
| **Cohérence** | Respect des contraintes | Solde bancaire jamais négatif |

| Propriété | Description | Exemple |
|---|---|---|
| **Isolation** | Indépendance des transactions concurrentes | Deux utilisateurs réservent le dernier siège |
| **Durabilité** | Persistance après COMMIT | Données préservées même en cas de panne |

> Analogie : Transaction bancaire
>
> Imaginez un virement de 100€ de A vers B :
>
> ```
> BEGIN;
> UPDATE comptes SET solde = solde - 100 WHERE titulaire = 'A';
> UPDATE comptes SET solde = solde + 100 WHERE titulaire = 'B';
> COMMIT;
> ```
>
> **Sans transaction** : Si le système plante entre les deux UPDATE, A perd 100€ mais B ne reçoit rien !
>
> **Avec transaction** : Soit les deux UPDATE réussissent, soit aucun.

## 8.2 Commandes de base

### 8.2.a BEGIN, COMMIT, ROLLBACK

```
-- Démarrer une transaction
BEGIN;

-- Opérations SQL
INSERT INTO etudiants (nom, prenom, email)
VALUES ('Test', 'Transaction', 'test@email.fr');

-- Valider les modifications
COMMIT;
```

```
-- Transaction avec annulation
BEGIN;

INSERT INTO cours (code, nom, credits)
VALUES ('IF999', 'Cours test', 3);
```

```sql
-- Annuler toutes les modifications
ROLLBACK;

-- Vérification : le cours IF999 n'existe pas
SELECT * FROM cours WHERE code = 'IF999';
```

**Résultat** : Aucune ligne (le ROLLBACK a annulé l'INSERT)

> ! 🔍 Comportement par défaut PostgreSQL
>
> PostgreSQL fonctionne en **mode auto-commit** :
>
> - Chaque instruction SQL = 1 transaction automatique
> - Équivalent à `BEGIN; instruction; COMMIT;`
>
> **Pour grouper des opérations** : Utiliser explicitement `BEGIN ... COMMIT`

## 8.3 Exemple pratique : inscription cohérente

### 8.3.a Problème sans transaction

Imaginez l'inscription d'un étudiant à plusieurs cours :

```sql
-- ✗ DANGEREUX : Sans transaction
INSERT INTO inscriptions VALUES (1, 4, NULL);  -- Cours 1
-- [PANNE SYSTÈME ICI]
INSERT INTO inscriptions VALUES (1, 5, NULL);  -- Cours 2
```

**Risque** : L'étudiant est inscrit au cours 1 mais pas au cours 2 → incohérence !

### 8.3.b Solution avec transaction

```sql
-- ☑ SÉCURISÉ : Avec transaction
BEGIN;

-- Inscription à plusieurs cours d'un coup
INSERT INTO inscriptions (etudiant_id, cours_id) VALUES
    (1, 4),
    (1, 5);

-- Vérification avant validation
SELECT c.nom
FROM cours c
JOIN inscriptions i ON c.id = i.cours_id
```

```
WHERE i.etudiant_id = 1;

COMMIT;
```

**Garantie** : Soit l'étudiant est inscrit aux deux cours, soit à aucun.

## 8.4 Savepoints : points de sauvegarde

Les **savepoints** permettent de créer des points de restauration intermédiaires.

```
BEGIN;

-- Première opération
INSERT INTO etudiants (nom, prenom, email)
VALUES ('Savepoint', 'Test', 'savepoint@email.fr');

-- Créer un point de sauvegarde
SAVEPOINT avant_cours;

-- Deuxième opération
INSERT INTO cours (code, nom, credits)
VALUES ('IF888', 'Cours savepoint', 4);

-- Erreur détectée : on annule seulement la dernière opération
ROLLBACK TO SAVEPOINT avant_cours;

-- L'étudiant est conservé, mais pas le cours
COMMIT;

-- Vérification
SELECT * FROM etudiants WHERE email = 'savepoint@email.fr';  -- Existe
SELECT * FROM cours WHERE code = 'IF888';  -- N'existe pas
```

**Utilité** : Annulation partielle sans perdre tout le travail.

> 💡 Cas d'usage des savepoints
>
> - **Imports de données** : Sauvegarder après chaque lot, rollback si erreur
> - **Procédures complexes** : Points de retour en arrière intermédiaires
> - **Gestion d'erreurs** : Capture d'exceptions dans PL/pgSQL

## 8.5 Niveaux d’isolation : comprendre la concurrence

### 8.5.a Problématique : anomalies de lecture

Quand plusieurs transactions s’exécutent **simultanément**, des problèmes peuvent survenir :

## Anomalies de concurrence

**1. Lecture sale (Dirty Read)**

Transaction `T1` lit des données **non validées** de `T2`.

```
T1: BEGIN                     T2: BEGIN
T1:                           T2: UPDATE etudiants SET nom='X' WHERE id=1
T1: SELECT nom FROM etudiants WHERE id=1  -- Lit 'X' (non committé)
T1:                           T2: ROLLBACK
T1: COMMIT
```

Problème : `T1` a lu une valeur qui n'existe plus !

**2. Lecture non reproductible (Non-Repeatable Read)**

Transaction `T1` lit deux fois la même donnée → résultats différents.

```
T1: BEGIN
T1: SELECT nom FROM etudiants WHERE id=1  -- Lit 'Dupont'
T1:                           T2: BEGIN
T1:                         T2: UPDATE etudiants SET nom='Martin' WHERE id=1
T1:                           T2: COMMIT
T1: SELECT nom FROM etudiants WHERE id=1  -- Lit 'Martin'
T1: COMMIT
```

Problème : Incohérence dans la même transaction.

**3. Lecture fantôme (Phantom Read)**

Transaction T1 exécute deux fois la même requête → nombre de lignes différent.

```
T1: BEGIN
T1: SELECT COUNT(*) FROM etudiants WHERE nom='Dupont'  -- Retourne 2
T1:                           T2: BEGIN
T1:                     T2: INSERT INTO etudiants VALUES (..., 'Dupont', ...)
T1:                           T2: COMMIT
T1: SELECT COUNT(*) FROM etudiants WHERE nom='Dupont'  -- Retourne 3
T1: COMMIT
```

Problème : Apparition de nouvelles lignes.

**4. Anomalie de sérialisation**

Deux transactions concurrentes produisent un résultat impossible si exécutées séquentiellement.

```
Compte A: 100€, Compte B: 100€

T1: BEGIN                     T2: BEGIN
T1: SELECT solde FROM A       T2: SELECT solde FROM B
T1: (lit 100)                 T2: (lit 100)
T1:                           T2: UPDATE A SET solde = solde + 100
T1:                           T2: COMMIT
```

### 8.5.b Les 4 niveaux d'isolation SQL

PostgreSQL implémente 4 niveaux d'isolation (standard SQL) :

| Niveau | Dirty Read | Non-Repeatable | Phantom | Sérialisation |
|---|---|---|---|---|
| **READ UNCOMMITTED** | ✓ Possible | ✓ Possible | ✓ Possible | ✓ Possible |
| **READ COMMITTED** | × Impossible | ✓ Possible | ✓ Possible | ✓ Possible |
| **REPEATABLE READ** | × Impossible | × Impossible | ✓ Possible* | ✓ Possible |
| **SERIALIZABLE** | × Impossible | × Impossible | × Impossible | × Impossible |

**PostgreSQL évite les phantoms en REPEATABLE READ (au-delà du standard SQL)**

> !🔍 Niveau par défaut PostgreSQL
>
> **READ COMMITTED** est le niveau par défaut.
>
> **Compromis** :
>
> - 🚦 Évite les lectures sales
> - 🚦 Bonnes performances
> - ⚠️ Permet lectures non reproductibles et fantômes

### 8.5.c Démonstration des niveaux d'isolation

Essayez les exemples suivants dans des sessions PostgreSQL distinctes pour observer les comportements.

#### 8.5.c.a READ COMMITTED (défaut)

Ouvrez **deux sessions PostgreSQL** distinctes :

**Session 1** :

```
BEGIN;
SELECT nom, prenom FROM etudiants WHERE id = 1;
-- Affiche : Dupont, Alice
-- [Laisser la transaction ouverte]
```

**Session 2** :

```
BEGIN;
UPDATE etudiants SET prenom = 'Alicia' WHERE id = 1;
COMMIT;
```

**Session 1** (suite) :

```
-- Toujours dans la même transaction
SELECT nom, prenom FROM etudiants WHERE id = 1;
-- Affiche : Dupont, Alicia (⚠ valeur différente)
COMMIT;
```

**Observation** : Lecture non reproductible !

#### 8.5.c.b REPEATABLE READ

**Session 1** :

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT nom, prenom FROM etudiants WHERE id = 1;
-- Affiche : Dupont, Alice
```

**Session 2** :

```
BEGIN;
UPDATE etudiants SET prenom = 'Alexandra' WHERE id = 1;
COMMIT;
```

**Session 1** (suite) :

```
SELECT nom, prenom FROM etudiants WHERE id = 1;
-- Affiche toujours : Dupont, Alice (snapshot figé)
COMMIT;

-- Maintenant, hors transaction :
SELECT nom, prenom FROM etudiants WHERE id = 1;
-- Affiche : Dupont, Alexandra
```

**Observation** : Les données lues sont **figées** au début de la transaction.

#### 8.5.c.c SERIALIZABLE

**Session 1** :

```sql
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT SUM(note) FROM inscriptions WHERE etudiant_id = 1;
-- Suppose que ça retourne 45.5
```

**Session 2** :

```sql
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT SUM(note) FROM inscriptions WHERE etudiant_id = 1;
-- Retourne aussi 45.5

-- Ajout d'une inscription
INSERT INTO inscriptions VALUES (1, 4, 15.0);
COMMIT;  -- ☑ Réussit
```

**Session 1** (suite) :

```sql
-- Tente de modifier les inscriptions
UPDATE inscriptions SET note = note + 1 WHERE etudiant_id = 1;
COMMIT;  -- ✕ ERREUR : could not serialize access due to read/write dependencies
```

**Observation** : PostgreSQL détecte le conflit de sérialisation et **annule** la transaction.

⚠️ ⚠ Gestion des erreurs SERIALIZABLE

Avec SERIALIZABLE, votre code applicatif doit **réessayer** les transactions échouées :

```python
import psycopg2

max_retries = 3
for attempt in range(max_retries):
    try:
        with conn.cursor() as cur:
            cur.execute("BEGIN ISOLATION LEVEL SERIALIZABLE")
            # ... opérations ...
            cur.execute("COMMIT")
        break  # Succès
    except psycopg2.extensions.TransactionRollbackError:
        if attempt == max_retries - 1:
            raise
        # Attendre avant de réessayer
        time.sleep(0.1 * (2 ** attempt))
```

## 8.6 Verrous (Locks) : contrôle explicite

### 8.6.a Types de verrous

PostgreSQL utilise des **verrous** pour gérer la concurrence :

| Verrou | Commande | Effet |
|---|---|---|
| **Row lock** | `SELECT ... FOR UPDATE` | Verrouille les lignes sélectionnées |
| **Table lock** | `LOCK TABLE` | Verrouille toute une table |
| **Advisory lock** | `pg_advisory_lock()` | Verrou personnalisé applicatif |

### 8.6.b SELECT FOR UPDATE

```
-- Session 1 : Réserver une place d'examen
BEGIN;

SELECT * FROM inscriptions
WHERE etudiant_id = 1 AND cours_id = 1
FOR UPDATE;  -- Verrouille cette inscription

-- [Traitement long...]

UPDATE inscriptions SET note = 18.0
WHERE etudiant_id = 1 AND cours_id = 1;

COMMIT;  -- Libère le verrou
```

**Session 2** (pendant que Session 1 est ouverte) :

```
BEGIN;

-- Tente de verrouiller la même ligne
SELECT * FROM inscriptions
WHERE etudiant_id = 1 AND cours_id = 1
FOR UPDATE;

-- ⧖ BLOQUÉ jusqu'au COMMIT de Session 1
```

**Utilité** : Éviter les conditions de concurrence (race conditions).

### 8.6.c FOR UPDATE vs FOR SHARE

| Mode | Lecture | Écriture | Compatible avec |
|---|---|---|---|
| **FOR UPDATE** | ✅ | ✅ Exclusif | Aucun autre FOR UPDATE/SHARE |
| **FOR SHARE** | ✅ | ❌ | Autres FOR SHARE |
| **FOR KEY SHARE** | ✅ | ❌ (sauf clés étrangères) | Autres FOR KEY SHARE |
| **FOR NO KEY UPDATE** | ✅ | ✅ (sauf clés primaires) | FOR KEY SHARE uniquement |

**Exemple FOR SHARE** :

```
-- Plusieurs sessions peuvent lire, aucune ne peut modifier
BEGIN;
SELECT * FROM etudiants WHERE id = 1 FOR SHARE;
-- [Lecture sécurisée sans bloquer d'autres lectures]
COMMIT;
```

### 8.6.d Deadlocks : interblocages

Un **deadlock** survient quand deux transactions s'attendent mutuellement.

**Session 1** :

```
BEGIN;
UPDATE etudiants SET nom = 'X' WHERE id = 1;  -- Verrouille ligne 1
-- [Attend...]
UPDATE etudiants SET nom = 'Y' WHERE id = 2;  -- Tente de verrouiller ligne 2
```

**Session 2** (simultanément) :

```
BEGIN;
UPDATE etudiants SET nom = 'A' WHERE id = 2;  -- Verrouille ligne 2
-- [Attend...]
UPDATE etudiants SET nom = 'B' WHERE id = 1;  -- Tente de verrouiller ligne 1
```

**Résultat** : 🔒 **DEADLOCK** détecté par PostgreSQL après quelques secondes.

**Réponse PostgreSQL** :

```
ERROR:  deadlock detected
DETAIL:  Process 1234 waits for ShareLock on transaction 5678
```

PostgreSQL **annule automatiquement** l'une des transactions (victime choisie aléatoirement).

> 💡 Prévenir les deadlocks
>
> **Bonnes pratiques** :
>
> 1. **Ordre fixe** : Toujours verrouiller les ressources dans le même ordre
>
>    ```
>    -- ☐ BON : Ordre croissant des IDs
>    UPDATE etudiants SET ... WHERE id = 1;
>    UPDATE etudiants SET ... WHERE id = 2;
>    ```
>
> 2. **Transactions courtes** : Minimiser le temps de verrouillage
> 3. **Utiliser NOWAIT** :
>
>    ```
>    SELECT ... FOR UPDATE NOWAIT;  -- Échoue immédiatement si verrouillé
>    ```
>
> 4. **Réessayer** : Implémenter une logique de retry en cas de deadlock

## 8.7 Patterns de concurrence courants

### 8.7.a Pattern 1 : Compteur incrémental thread-safe

**✗ INCORRECT** (race condition) :

```
-- Session 1 et 2 simultanément
SELECT count FROM compteur WHERE id = 1;  -- Les deux lisent 100
-- [Les deux calculent 101]
UPDATE compteur SET count = 101 WHERE id = 1;
-- Résultat : 101 au lieu de 102
```

**✓ CORRECT** :

```
-- Utiliser FOR UPDATE
BEGIN;
SELECT count FROM compteur WHERE id = 1 FOR UPDATE;
UPDATE compteur SET count = count + 1 WHERE id = 1;
COMMIT;
```

**OU** (plus simple) :

```
-- Mise à jour atomique directe
UPDATE compteur SET count = count + 1 WHERE id = 1 RETURNING count;
```

### 8.7.b Pattern 2 : Places limitées (overbooking)

Gérer un cours avec **places limitées** :

```
-- Table des cours avec capacité
ALTER TABLE cours ADD COLUMN places_max INTEGER;
ALTER TABLE cours ADD COLUMN places_occupees INTEGER DEFAULT 0;

UPDATE cours SET places_max = 30, places_occupees = 0 WHERE id = 1;
```

**Inscription sécurisée** :

```
BEGIN ISOLATION LEVEL SERIALIZABLE;

-- Vérifier la disponibilité
SELECT places_max, places_occupees
FROM cours
WHERE id = 1
FOR UPDATE;  -- Verrouille la ligne

-- Si des places disponibles :
INSERT INTO inscriptions (etudiant_id, cours_id) VALUES (5, 1);

-- Incrémenter le compteur
UPDATE cours
SET places_occupees = places_occupees + 1
WHERE id = 1
  AND places_occupees < places_max;  -- Double vérification

-- Vérifier que l'update a réussi
IF FOUND THEN
    COMMIT;
ELSE
    ROLLBACK;  -- Plus de places
END IF;
```

**Version PL/pgSQL complète** :

```
CREATE OR REPLACE FUNCTION inscrire_etudiant(p_etudiant_id INTEGER, p_cours_id
INTEGER)
RETURNS BOOLEAN AS $$
```

```
DECLARE
    v_places_dispo INTEGER;
BEGIN
    -- Verrouiller le cours
    SELECT places_max - places_occupees INTO v_places_dispo
    FROM cours
    WHERE id = p_cours_id
    FOR UPDATE;

    -- Vérifier disponibilité
    IF v_places_dispo ⩽ 0 THEN
        RETURN FALSE;
    END IF;

    -- Inscrire
    INSERT INTO inscriptions (etudiant_id, cours_id)
    VALUES (p_etudiant_id, p_cours_id);

    -- Incrémenter
    UPDATE cours
    SET places_occupees = places_occupees + 1
    WHERE id = p_cours_id;

    RETURN TRUE;
END;
$$ LANGUAGE plpgsql;

-- Utilisation
BEGIN;
SELECT inscrire_etudiant(5, 1);  -- Retourne TRUE ou FALSE
COMMIT;
```

### 8.7.c Pattern 3 : Optimistic Locking (version)

Au lieu de verrouiller, utiliser un **numéro de version** :

```
-- Ajouter une colonne version
ALTER TABLE etudiants ADD COLUMN version INTEGER DEFAULT 0;

-- Lecture
BEGIN;
SELECT id, nom, prenom, version FROM etudiants WHERE id = 1;
-- Suppose : version = 5
```

**Modification avec vérification** :

```sql
UPDATE etudiants
SET nom = 'Nouveau Nom',
    version = version + 1
WHERE id = 1
  AND version = 5;  -- ⚠ Vérifie la version

-- Vérifier le nombre de lignes affectées
GET DIAGNOSTICS v_updated = ROW_COUNT;

IF v_updated = 0 THEN
    ROLLBACK;  -- Conflit détecté
      RAISE EXCEPTION 'Conflit de version : données modifiées par un autre
utilisateur';
ELSE
    COMMIT;
END IF;
```

**Avantage** : Pas de verrous, détection de conflits post-facto.

## 8.8 Exercices — Transactions et isolation

### ℹ️ Exercice 8.1 — Virement bancaire

Créer un système de **virement bancaire** sécurisé.

**Schéma** :

```sql
CREATE TABLE comptes (
    id SERIAL PRIMARY KEY,
    titulaire VARCHAR(100),
    solde NUMERIC(10,2) CHECK (solde >= 0)
);

INSERT INTO comptes (titulaire, solde) VALUES
    ('Alice', 1000.00),
    ('Bob', 500.00);
```

**Travail** :

1. Écrire une fonction `virement(source_id, dest_id, montant)` qui :
   - Débite le compte source
   - Crédite le compte destination
   - Garantit l'atomicité
   - Vérifie que le solde source reste ≥ 0
2. Tester avec deux sessions simultanées essayant de virer depuis le même compte

#### Solution

```sql
CREATE OR REPLACE FUNCTION virement(
    p_source_id INTEGER,
    p_dest_id INTEGER,
    p_montant NUMERIC
)
RETURNS BOOLEAN AS $$
DECLARE
    v_solde_source NUMERIC;
BEGIN
   -- Verrouiller le compte source (ordre important : toujours même ordre d'ID)
    SELECT solde INTO v_solde_source
    FROM comptes
    WHERE id = LEAST(p_source_id, p_dest_id)
    FOR UPDATE;

    -- Verrouiller le compte destination
    IF p_source_id <> p_dest_id THEN
        PERFORM 1 FROM comptes WHERE id = GREATEST(p_source_id, p_dest_id)
FOR UPDATE;
    END IF;

    -- Vérifier le solde
    SELECT solde INTO v_solde_source FROM comptes WHERE id = p_source_id;
```

### ℹ️ Exercice 8.2 — Réservation de siège

Simuler un système de réservation de **sièges de cinéma**.

**Schéma** :

```
CREATE TABLE seances (
    id SERIAL PRIMARY KEY,
    film VARCHAR(100),
    horaire TIMESTAMP,
    places_total INTEGER,
    places_reservees INTEGER DEFAULT 0
);

CREATE TABLE reservations (
    id SERIAL PRIMARY KEY,
    seance_id INTEGER REFERENCES seances(id),
    nom_client VARCHAR(100),
    nb_places INTEGER,
    timestamp_reservation TIMESTAMP DEFAULT NOW()
);

INSERT INTO seances (film, horaire, places_total) VALUES
    ('Matrix', '2024-02-15 20:00', 100);
```

**Travail** :

1. Écrire une fonction `reserver_places(seance_id, nom, nb_places)` qui :
   - Vérifie la disponibilité
   - Empêche l'overbooking (même avec concurrence)
   - Crée la réservation
   - Met à jour le compteur
2. Tester avec 10 sessions simultanées réservant les dernières places
3. Comparer les comportements en READ COMMITTED vs SERIALIZABLE

▌Solution

```
CREATE OR REPLACE FUNCTION reserver_places(
    p_seance_id INTEGER,
    p_nom VARCHAR,
    p_nb_places INTEGER
)
RETURNS INTEGER AS $$
DECLARE
    v_places_dispo INTEGER;
    v_reservation_id INTEGER;
BEGIN
    -- Verrouiller la séance
    SELECT places_total - places_reservees INTO v_places_dispo
    FROM seances
```

ℹ️ Exercice 8.3 — Détection d'anomalies

Observer les différentes anomalies de concurrence.

**Travail** :

1. **Lecture non reproductible** :
   - Session 1 : Lire une note en READ COMMITTED
   - Session 2 : Modifier cette note
   - Session 1 : Relire → observer le changement
2. **Phantom read** :
   - Session 1 : Compter les inscriptions d'un étudiant
   - Session 2 : Ajouter une inscription
   - Session 1 : Recompter → observer la différence
3. **Serialization anomaly** :
   - Reproduire l'exemple du virement croisé
   - Tester en SERIALIZABLE → observer l'erreur

💡 Indice Utilisez deux fenêtres `psql` ou deux sessions dans pgAdmin.

Commandes utiles :

```
-- Changer le niveau d'isolation
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Voir le niveau actuel
SHOW transaction_isolation;
```

## 8.9 Bonnes pratiques et recommandations

### 8.9.a Choix du niveau d'isolation

| Cas d'usage | Niveau recommandé | Justification |
|---|---|---|
| Lecture seule (rapports) | READ COMMITTED | Performances optimales |
| Modifications simples (CRUD) | READ COMMITTED | Compromis perf/cohérence |
| Calculs critiques (comptabilité) | REPEATABLE READ | Cohérence des lectures |
| Logique métier complexe | SERIALIZABLE | Garantie absolue |

### 8.9.b Règles d'or

1. **Transactions courtes** : Minimiser la durée
2. **Ordre fixe** : Toujours accéder aux ressources dans le même ordre
3. **Gestion d'erreurs** : Toujours prévoir ROLLBACK
4. **Retry logic** : Réessayer les transactions SERIALIZABLE échouées
5. **Éviter SELECT * FOR UPDATE** : Verrouiller uniquement ce qui est nécessaire
6. **Surveiller les deadlocks** : Logs PostgreSQL (`log_lock_waits`)

### 8.9.c Monitoring

```
-- Voir les transactions actives
SELECT pid, usename, state, query_start, query
FROM pg_stat_activity
WHERE state <> 'idle';

-- Voir les verrous
SELECT locktype, relation::regclass, mode, granted
FROM pg_locks
WHERE NOT granted;  -- Verrous en attente

-- Configuration du timeout
SET lock_timeout = '5s';  -- Annule après 5s d'attente
SET statement_timeout = '30s';  -- Annule après 30s d'exécution
```

## 8.10 Conclusion

Les transactions et l'isolation sont **essentielles** pour garantir la cohérence des données dans un environnement multi-utilisateurs.

**Points clés à retenir** :

**ACID** : Atomicité, Cohérence, Isolation, Durabilité

**4 niveaux** : READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE

**Verrous** : FOR UPDATE, FOR SHARE pour contrôler explicitement

**Deadlocks** : Inévitables, mais gérables (ordre fixe, retry)

**Performance vs Cohérence** : Choisir le bon niveau d'isolation

**Prochaine étape** : Appliquer ces concepts dans vos projets réels, notamment les applications web avec concurrence élevée.

```
#| echo: false
#| output: false
```

```
# Nettoyage de fin de leçon (même garde que le balayage de la leçon 00) :
# le conteneur `db` lancé ici retient le port 5432 de l'hôte. Sans libération
# en fin de page, il bloque le démarrage du PostgreSQL des pages suivantes du
# même pod de rendu (ex. leçon JDBC).
if [ -n "$CI" ]; then
  docker rm -f db >/dev/null 2>&1 || true
fi
```