Bases de Données

Notes de Cours

Licence L3 — 2025–2026

Yaë Ulrich Gaba

« Les données sont le nouveau pétrole. »
— Clive Humby

25 mars 2026

Table des matières

Préface **1**

1 Introduction aux Bases de Données Relationnelles **7**
1.1 Qu'est-ce qu'une base de données ? . . . 7
1.2 Bref historique . . . 8
1.3 Système de Gestion de Bases de Données (SGBD) . . . 8
1.3.1 Architecture trois niveaux (ANSI/SPARC) . . . 8
1.4 Le modèle relationnel . . . 9
1.5 Langages d'un SGBD . . . 10
1.5.1 DDL — Langage de Définition de Données . . . 10
1.5.2 DML — Langage de Manipulation de Données . . . 10
1.5.3 DCL — Langage de Contrôle de Données . . . 11
1.6 Principaux SGBD relationnels . . . 11
1.7 Premiers pas avec SQLite et Python . . . 11
1.8 Types de données SQL . . . 12
1.9 Contraintes d'intégrité . . . 13
1.10 Modèles de données : panorama . . . 13

2 Le Modèle Entité-Relation **15**
2.1 Introduction à la conception de bases de données . . . 15
2.2 Concepts fondamentaux . . . 15
2.3 Cardinalités . . . 16
2.4 Diagrammes E/R avec TikZ . . . 16
2.5 Entités faibles . . . 17
2.6 Spécialisation et généralisation . . . 17
2.7 Passage du modèle E/R au modèle relationnel . . . 17
2.8 Exemple complet : système de bibliothèque . . . 18
2.8.1 Analyse des besoins . . . 18
2.8.2 Diagramme E/R . . . 19
2.8.3 Traduction en SQL . . . 19
2.9 Bonnes pratiques de conception . . . 20
2.10 Intégration Python : génération de schéma . . . 20

3 Algèbre Relationnelle **23**
3.1 Introduction . . . 23
3.2 Sélection (σ) . . . 23
3.3 Projection (π) . . . 24
3.4 Opérateurs ensemblistes . . . 25
3.5 Produit cartésien ($\times$) . . . 25

3.6 Jointure ($\bowtie$) . . . 26
3.7 Renommage (ρ) . . . 26
3.8 Division ($\div$) . . . 27
3.9 Arbre algébrique et optimisation . . . 27
3.10 Résumé des correspondances algèbre–SQL . . . 28
3.11 Intégration Python . . . 28

4 SQL — Requêtes de Base 31
4.1 Introduction à SQL . . . 31
4.2 Structure d'une requête SELECT . . . 31
4.3 SELECT et FROM . . . 32
4.4 Alias de colonnes et de tables . . . 32
4.5 Clause WHERE — Filtrage . . . 33
4.5.1 Opérateurs de comparaison . . . 33
4.5.2 Opérateurs logiques . . . 33
4.5.3 LIKE et les motifs . . . 34
4.6 ORDER BY — Tri . . . 34
4.7 DISTINCT — Élimination des doublons . . . 35
4.8 LIMIT et OFFSET — Pagination . . . 35
4.9 Expressions et fonctions scalaires . . . 36
4.10 Fonctions utiles . . . 37
4.11 INSERT, UPDATE, DELETE . . . 37
4.12 Intégration Python complète . . . 38

5 SQL — Jointures et Sous-requêtes 41
5.1 Introduction . . . 41
5.2 Jointure interne (INNER JOIN) . . . 41
5.3 Jointure naturelle (NATURAL JOIN) . . . 42
5.4 Jointures externes (OUTER JOIN) . . . 42
5.4.1 LEFT JOIN . . . 43
5.4.2 RIGHT JOIN . . . 43
5.4.3 FULL OUTER JOIN . . . 43
5.5 Jointure croisée (CROSS JOIN) . . . 44
5.6 Auto-jointure (Self Join) . . . 44
5.7 Sous-requêtes . . . 45
5.7.1 Sous-requête scalaire . . . 45
5.7.2 Sous-requête avec IN . . . 45
5.7.3 Sous-requête corrélée . . . 45
5.7.4 Sous-requête dans FROM (table dérivée) . . . 46
5.7.5 Sous-requête dans SELECT . . . 47
5.8 Opérateurs ALL, ANY/SOME . . . 47
5.9 Expressions de table communes (CTE) . . . 48
5.10 Intégration Python . . . 49

6 SQL — Agrégation et Fonctions de Fenêtre 51
6.1 Fonctions d'agrégation . . . 51
6.2 GROUP BY — Regroupement . . . 52
6.3 HAVING — Filtrage après regroupement . . . 53
6.4 Groupements avancés . . . 54

6.4.1 GROUPING SETS, ROLLUP, CUBE 54
6.5 Fonctions de fenêtre (Window Functions) 54
6.5.1 Syntaxe OVER 55
6.5.2 Agrégation fenêtrée 55
6.5.3 ROW_NUMBER, RANK, DENSE_RANK 55
6.5.4 LAG, LEAD 56
6.5.5 Fenêtres glissantes (Frame) 57
6.5.6 NTILE et calculs de pourcentage 57
6.6 Intégration Python 57

7 Normalisation des Bases de Données **59**
7.1 Problèmes de conception et anomalies 59
7.2 Dépendances fonctionnelles 60
7.3 Axiomes d'Armstrong 60
7.4 Fermeture d'un ensemble d'attributs 60
7.5 Première forme normale (1NF) 61
7.6 Deuxième forme normale (2NF) 61
7.7 Troisième forme normale (3NF) 62
7.8 Forme normale de Boyce-Codd (BCNF) 62
7.9 Décomposition sans perte 62
7.10 Algorithme de décomposition en BCNF 63
7.11 Résumé des formes normales 63
7.12 Synthèse 3NF (algorithme de Bernstein) 63
7.13 Exemple complet de normalisation 64
7.14 Dénormalisation contrôlée 65

8 Transactions et Contrôle de Concurrence **67**
8.1 Notion de transaction 67
8.2 Propriétés ACID 68
8.3 Problèmes liés à la concurrence 68
8.4 Niveaux d'isolation 68
8.5 Théorie de la sérialisabilité 69
8.6 Verrouillage à deux phases (2PL) 69
8.7 Détection et prévention des interblocages 70
8.8 Contrôle de concurrence multi-versions (MVCC) 70
8.9 Journal de transactions (WAL) 70
8.10 Formules clés 71
8.11 Exercices 71

9 Indexation et Structures de Données **73**
9.1 Motivation 73
9.2 Arbres B et B^+ 73
9.3 Index de hachage 74
9.4 Index bitmap 74
9.5 Index spatiaux : R-tree 75
9.6 Index plein texte 75
9.7 Stratégies de sélection d'index 75
9.8 Formules clés 76
9.9 Exercices 76

10 ETL et Entrepôts de Données **79**
10.1 Concepts fondamentaux . . . 79
10.2 Modélisation dimensionnelle . . . 80
10.3 Processus ETL . . . 80
10.4 Dimensions à évolution lente (SCD) . . . 81
10.5 Opérations OLAP . . . 82
10.6 Vues matérialisées . . . 82
10.7 Formules clés . . . 83
10.8 Exercices . . . 83

11 Bases de Données NoSQL **85**
11.1 Motivation et contexte . . . 85
11.2 Théorème CAP . . . 85
11.3 Bases clé-valeur . . . 86
11.4 Bases document . . . 86
11.5 Bases orientées colonnes . . . 87
11.6 Bases de données orientées graphe . . . 88
11.7 Comparaison et choix . . . 88
11.8 Formules clés . . . 89
11.9 Exercices . . . 89

Préface

Objectifs de ce cours

Les bases de données constituent l'un des piliers fondamentaux de l'informatique moderne. Depuis les premiers systèmes de gestion de fichiers des années 1960 jusqu'à l'explosion des bases NoSQL et des lacs de données (*data lakes*), la capacité à stocker, organiser et interroger des données de manière fiable et performante reste un enjeu central pour toute application informatique.

Ce cours s'adresse aux étudiants de deuxième année de licence (L2) en informatique, mathématiques-informatique ou sciences des données. Il suppose une familiarité de base avec la programmation (Python) et les mathématiques discrètes (ensembles, relations, logique propositionnelle).

À l'issue de ce cours, l'étudiant sera capable de :

- Concevoir un schéma conceptuel (modèle Entité-Relation) et le traduire en schéma relationnel normalisé.
- Écrire des requêtes SQL complexes impliquant jointures, agrégations, sous-requêtes et fonctions fenêtres.
- Comprendre et appliquer les formes normales (1NF à BCNF) pour éliminer les anomalies de mise à jour.
- Expliquer les propriétés ACID et les mécanismes de contrôle de concurrence.
- Choisir et créer des index adaptés pour optimiser les performances.
- Concevoir un pipeline ETL simple avec Python et SQL.
- Comparer les modèles NoSQL (document, clé-valeur, colonne, graphe) au modèle relationnel.

Organisation du cours

Le cours est structuré en onze chapitres, progressant des fondements théoriques vers les applications pratiques :

1. **Introduction aux bases de données relationnelles** — Histoire, concepts fondamentaux, architecture d'un SGBD.
2. **Modèle Entité-Relation** — Conception de schémas conceptuels, diagrammes E/R, passage au relationnel.

3. **Algèbre relationnelle** — Opérateurs formels : sélection, projection, jointure, division.
4. **SQL — Requêtes de base** — `SELECT`, `FROM`, `WHERE`, tri et filtrage.
5. **SQL — Jointures et sous-requêtes** — Jointures internes, externes, croisées ; sous-requêtes corrélées.
6. **SQL — Agrégation et fonctions fenêtres** — `GROUP BY`, `HAVING`, `ROW_NUMBER`, `RANK`.
7. **Normalisation** — Dépendances fonctionnelles, 1NF à BCNF, décomposition sans perte.
8. **Transactions, intégrité, contraintes** — Propriétés ACID, niveaux d'isolation, verrouillage.
9. **Indexation et optimisation** — B-arbres, index de hachage, plans d'exécution.
10. **Pipelines de données et ETL** — Extraction, transformation, chargement ; intégration Python/SQL.
11. **Bases NoSQL** — Document, clé-valeur, colonne, graphe ; théorème CAP.

Approche pédagogique

Chaque chapitre suit une structure cohérente :

- **Théorie** — Définitions formelles, théorèmes et propositions avec démonstrations ou justifications.
- **Exemples détaillés** — Chaque concept est illustré par un ou plusieurs exemples concrets, souvent sur une base de données « Université »utilisée comme fil rouge.
- **Code SQL exécutable** — Les requêtes sont présentées avec leur résultat attendu, testables sur SQLite, PostgreSQL ou MySQL.
- **Intégration Python** — Des scripts `sqlite3` et `pandas` montrent l'utilisation programmatique.
- **Exercices** — Des exercices de difficulté croissante, allant de la vérification de compréhension à la résolution de problèmes ouverts.

Base de données fil rouge

Tout au long de ce cours, nous utilisons une base de données **Université** comportant les tables suivantes :

Schéma de la base Université

```
CREATE TABLE Etudiants (
```

```
    id_etudiant  INTEGER PRIMARY KEY,
    nom          TEXT NOT NULL,
    prenom       TEXT NOT NULL,
    date_naissance DATE,
    filiere      TEXT
);

CREATE TABLE Cours (
    id_cours     INTEGER PRIMARY KEY,
    intitule     TEXT NOT NULL,
    credits      INTEGER DEFAULT 3,
    id_prof      INTEGER REFERENCES Professeurs(id_prof)
);

CREATE TABLE Professeurs (
    id_prof      INTEGER PRIMARY KEY,
    nom          TEXT NOT NULL,
    prenom       TEXT NOT NULL,
    departement  TEXT
);

CREATE TABLE Inscriptions (
    id_etudiant  INTEGER REFERENCES Etudiants(id_etudiant),
    id_cours     INTEGER REFERENCES Cours(id_cours),
    note         REAL,
    annee        INTEGER,
    PRIMARY KEY (id_etudiant, id_cours)
);
```

Ce schéma sera enrichi au fil des chapitres pour illustrer des concepts avancés (héritage, relations ternaires, historisation).

Données d'exemple

Insertion des données d'exemple

```
INSERT INTO Professeurs VALUES (1, 'Dupont', 'Marie', 'Informatique');
INSERT INTO Professeurs VALUES (2, 'Martin', 'Jean', 'Mathematiques');
INSERT INTO Professeurs VALUES (3, 'Durand', 'Sophie', 'Informatique');

INSERT INTO Cours VALUES (101, 'Bases de donnees', 4, 1);
INSERT INTO Cours VALUES (102, 'Algorithmes', 3, 2);
INSERT INTO Cours VALUES (103, 'Reseaux', 3, 3);
INSERT INTO Cours VALUES (104, 'Statistiques', 3, 2);

INSERT INTO Etudiants VALUES (1, 'Bernard', 'Alice', '2004-03-15',
↪  'Info');
INSERT INTO Etudiants VALUES (2, 'Petit', 'Bob', '2003-11-22', 'Info');
```

```sql
INSERT INTO Etudiants VALUES (3, 'Moreau', 'Claire', '2004-07-08',
↪  'Maths');
INSERT INTO Etudiants VALUES (4, 'Leroy', 'David', '2003-01-30', 'Info');
INSERT INTO Etudiants VALUES (5, 'Roux', 'Emma', '2004-09-12', 'Maths');

INSERT INTO Inscriptions VALUES (1, 101, 15.5, 2025);
INSERT INTO Inscriptions VALUES (1, 102, 13.0, 2025);
INSERT INTO Inscriptions VALUES (2, 101, 12.0, 2025);
INSERT INTO Inscriptions VALUES (2, 103, 16.0, 2025);
INSERT INTO Inscriptions VALUES (3, 102, 17.5, 2025);
INSERT INTO Inscriptions VALUES (3, 104, 14.0, 2025);
INSERT INTO Inscriptions VALUES (4, 101, 9.0, 2025);
INSERT INTO Inscriptions VALUES (5, 104, 18.0, 2025);
```

Outils recommandés

- **SQLite** — Léger, sans serveur, idéal pour l'apprentissage. Disponible via `sqlite3` en ligne de commande ou le module Python `sqlite3`.
- **PostgreSQL** — SGBD complet, conforme au standard SQL, recommandé pour les exercices avancés (fonctions fenêtres, transactions concurrentes).
- **DB Browser for SQLite** — Interface graphique pour visualiser et manipuler des bases SQLite.
- **DBeaver** — Client universel supportant PostgreSQL, MySQL, SQLite et d'autres.
- **Python 3.x** — Avec les modules `sqlite3` (standard), `psycopg2` (PostgreSQL), `pandas` (analyse de données).

Conventions typographiques

- Les mots-clés SQL sont écrits en `MAJUSCULES` : `SELECT`, `WHERE`, `JOIN`.
- Les noms de tables et colonnes sont en `PascalCase` ou `snake_case` selon le contexte.
- Les termes techniques apparaissant pour la première fois sont en *italique*.
- Les encadrés colorés signalent :
  - les **définitions** et **théorèmes** importants,
  - les **bonnes pratiques** à retenir,
  - les **anti-patterns** à éviter,
  - les **avertissements** sur les pièges courants.

Prérequis

- **Programmation** — Savoir écrire un programme Python simple (variables, boucles, fonctions, fichiers).
- **Mathématiques** — Notions d'ensembles, de relations, de produit cartésien, de logique propositionnelle ($\wedge$, $\vee$, $\neg$, $\Rightarrow$).
- **Système** — Savoir utiliser un terminal (ligne de commande).

Ressources complémentaires

- A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, McGraw-Hill, 7^{e} édition.
- R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, Pearson, 7^{e} édition.
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, McGraw-Hill, 3^{e} édition.
- Documentation officielle de PostgreSQL : `https://www.postgresql.org/docs/`
- Documentation SQLite : `https://www.sqlite.org/docs.html`
- W3Schools SQL Tutorial : `https://www.w3schools.com/sql/`

Bonne lecture et bon apprentissage !

Chapitre 1

Introduction aux Bases de Données Relationnelles

En 1970, Edgar F. Codd, chercheur chez IBM, publie un article qui va révolutionner l'informatique : *A Relational Model of Data for Large Shared Data Banks*. Son idée : organiser les données dans des *tables* reliées par des opérations algébriques précises, en séparant radicalement la manière dont les données sont stockées physiquement de la manière dont on les interroge logiquement. IBM, ironie de l'histoire, met des années à implémenter les idées de son propre chercheur — c'est un groupe de Berkeley, mené par Michael Stonebraker, qui crée le premier système relationnel pratique. Aujourd'hui, les bases de données relationnelles stockent l'essentiel de l'information mondiale.

1.1 Qu'est-ce qu'une base de données ?

Commençons par la question la plus fondamentale.

Définition 1.1 (Base de données)**.** Une *base de données* est une collection organisée de données structurées, stockées de manière persistante et accessibles par plusieurs utilisateurs et applications de façon contrôlée.

Contrairement à un simple fichier (CSV, texte), une base de données offre :

- **Persistance** — les données survivent à l'arrêt du programme.
- **Partage** — plusieurs utilisateurs accèdent simultanément aux données.
- **Intégrité** — des contraintes garantissent la cohérence des données.
- **Sécurité** — des mécanismes de contrôle d'accès protègent les données.
- **Interrogation efficace** — un langage déclaratif (SQL) permet de poser des questions complexes sans décrire le « comment ».

Exemple 1.2. Considérons un système de gestion d'une université. Sans base de données, on pourrait stocker les étudiants dans un fichier CSV :

```
1,Bernard,Alice,2004-03-15,Info
2,Petit,Bob,2003-11-22,Info
```

Problèmes : pas de vérification de type, pas de clé étrangère, pas d'accès concurrent sûr, pas de requêtes complexes efficaces.

1.2 Bref historique

1. **Années 1960** — Systèmes de fichiers, modèles hiérarchiques (IMS d'IBM) et réseau (CODASYL). Les données sont organisées en arbres ou graphes avec navigation explicite.

2. **1970** — Edgar F. Codd publie *"A Relational Model of Data for Large Shared Data Banks"*, posant les bases du modèle relationnel. Les données sont organisées en *tables* (relations) manipulées par des opérateurs ensemblistes.

3. **Années 1970–80** — Développement de System R (IBM) et Ingres (Berkeley), naissance du langage SQL (initialement SEQUEL).

4. **Années 1980–90** — Commercialisation : Oracle, DB2, SQL Server, PostgreSQL. Standardisation de SQL (SQL-86, SQL-92, SQL :1999).

5. **Années 2000** — Bases de données orientées objets, XML, entrepôts de données (*data warehouses*).

6. **Années 2010+** — Explosion NoSQL (MongoDB, Cassandra, Neo4j), NewSQL, bases en mémoire, lacs de données.

Remarque 1.3. Malgré l'essor du NoSQL, le modèle relationnel reste le plus utilisé en entreprise. Les compétences SQL sont parmi les plus demandées sur le marché de l'emploi en informatique.

1.3 Système de Gestion de Bases de Données (SGBD)

Définition 1.4 (SGBD)**.** Un *Système de Gestion de Bases de Données* (SGBD, en anglais *DBMS*) est un logiciel qui permet de créer, gérer et interroger des bases de données. Il assure l'indépendance entre les programmes d'application et la représentation physique des données.

Un SGBD fournit les fonctions suivantes (règle de Codd) :

- **Définition des données** (DDL) — création de tables, contraintes, index.
- **Manipulation des données** (DML) — insertion, modification, suppression, interrogation.
- **Contrôle des données** (DCL) — droits d'accès, rôles.
- **Contrôle de transactions** (TCL) — `BEGIN`, `COMMIT`, `ROLLBACK`.

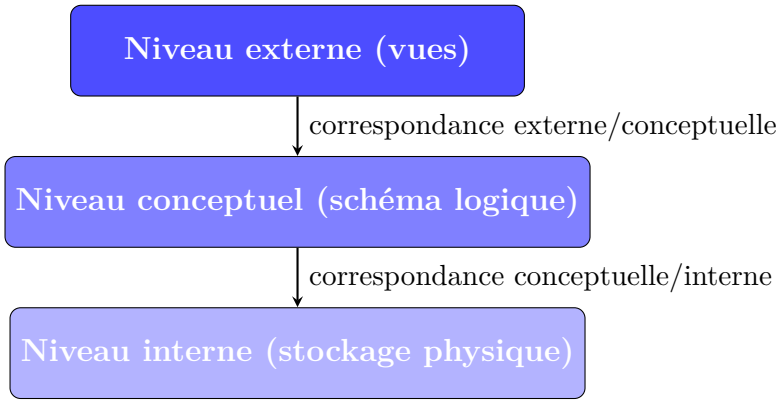
1.3.1 Architecture trois niveaux (ANSI/SPARC)

Définition 1.5 (Architecture ANSI/SPARC)**.** L'architecture ANSI/SPARC (1975) définit trois niveaux d'abstraction :

1. **Niveau externe** (vues utilisateur) — chaque utilisateur ou application voit un sous-ensemble adapté des données.

2. **Niveau conceptuel** (schéma logique) — description complète de la structure des données, indépendante du stockage physique.

3. **Niveau interne** (schéma physique) — organisation sur disque : fichiers, index, partitions.



Théorème 1.6 (Indépendance des données)**.** *L'architecture trois niveaux garantit deux formes d'indépendance :*

- ***Indépendance logique*** *: on peut modifier le schéma conceptuel sans affecter les vues externes.*
- ***Indépendance physique*** *: on peut modifier l'organisation physique (ajout d'index, changement de format) sans affecter le schéma conceptuel.*

1.4 Le modèle relationnel

Définition 1.7 (Relation)**.** Soit $D_1, D_2, \ldots, D_n$ des domaines (ensembles de valeurs possibles). Une *relation* R de schéma $R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$ est un sous-ensemble fini du produit cartésien $D_1 \times D_2 \times \cdots \times D_n$. Chaque élément de R est appelé un *n-uplet* (ou *tuple*).

Terminologie relationnelle vs. tabulaire

Modèle relationnel	Tableau	SQL
Relation	Table	`TABLE`
Attribut	Colonne	`COLUMN`
Tuple (n-uplet)	Ligne	`ROW`
Domaine	Type de données	`INTEGER`, `TEXT`, …
Degré	Nombre de colonnes	—
Cardinalité	Nombre de lignes	`COUNT(*)`

Définition 1.8 (Clé candidate et clé primaire)**.** Soit R une relation de schéma $R(A_1, \ldots, A_n)$.

- Une *super-clé* de R est un sous-ensemble $K \subseteq \{A_1, \ldots, A_n\}$ tel que pour tout couple de tuples distincts $t_1, t_2 \in R$, on a $t_1[K] \neq t_2[K]$.

- Une *clé candidate* est une super-clé minimale (aucun sous-ensemble strict n'est lui-même super-clé).
- La *clé primaire* est la clé candidate choisie par le concepteur pour identifier de manière unique chaque tuple.

Exemple 1.9. Dans la table `Etudiants` :

- {`id_etudiant`} est clé candidate (et clé primaire choisie).
- {`nom`, `prenom`, `date_naissance`} pourrait être une clé candidate si on suppose l'unicité de cette combinaison.
- {`id_etudiant`, `nom`} est une super-clé mais pas une clé candidate (non minimale).

Définition 1.10 (Clé étrangère)**.** Soit deux relations R et S. Un ensemble d'attributs FK de R est une *clé étrangère* référençant S si, pour tout tuple $t \in R$, $t[FK]$ est soit `NULL`, soit égal à $s[PK]$ pour un tuple $s \in S$, où PK est la clé primaire de S.

1.5 Langages d'un SGBD

1.5.1 DDL — Langage de Définition de Données

Création d'une table avec contraintes

```
CREATE TABLE Etudiants (
    id_etudiant  INTEGER PRIMARY KEY,
    nom          TEXT NOT NULL,
    prenom       TEXT NOT NULL,
    date_naissance DATE,
    filiere      TEXT CHECK (filiere IN ('Info', 'Maths', 'Physique')),
    UNIQUE (nom, prenom, date_naissance)
);
```

1.5.2 DML — Langage de Manipulation de Données

Opérations CRUD de base

```
-- Create (insertion)
INSERT INTO Etudiants (id_etudiant, nom, prenom, filiere)
VALUES (6, 'Garcia', 'Lucie', 'Info');

-- Read (lecture)
SELECT nom, prenom FROM Etudiants WHERE filiere = 'Info';

-- Update (mise a jour)
UPDATE Etudiants SET filiere = 'Maths' WHERE id_etudiant = 6;

-- Delete (suppression)
DELETE FROM Etudiants WHERE id_etudiant = 6;
```

1.5.3 DCL — Langage de Contrôle de Données

Gestion des droits (PostgreSQL)

```
-- Accorder le droit de lecture
GRANT SELECT ON Etudiants TO utilisateur_web;

-- Accorder tous les droits
GRANT ALL PRIVILEGES ON Cours TO administrateur;

-- Revoquer un droit
REVOKE DELETE ON Etudiants FROM utilisateur_web;
```

1.6 Principaux SGBD relationnels

SGBD	Licence	Usage typique	Particularité
PostgreSQL	Open source	Entreprise, web	Très conforme SQL
MySQL/MariaDB	Open source	Web (LAMP)	Très répandu
SQLite	Domaine public	Embarqué, mobile	Sans serveur
Oracle DB	Commercial	Grande entreprise	Très performant
SQL Server	Commercial	Entreprise (.NET)	Intégration Microsoft

1.7 Premiers pas avec SQLite et Python

SQLite est intégré à Python via le module `sqlite3` :

Connexion et création de table avec Python

```
import sqlite3

# Connexion (cree le fichier si inexistant)
conn = sqlite3.connect('universite.db')
cur = conn.cursor()

# Creation de la table
cur.execute('''
    CREATE TABLE IF NOT EXISTS Etudiants (
        id_etudiant  INTEGER PRIMARY KEY,
        nom          TEXT NOT NULL,
        prenom       TEXT NOT NULL,
        date_naissance DATE,
        filiere      TEXT
    )
''')

# Insertion
```

```
cur.execute("INSERT INTO Etudiants VALUES (1, 'Bernard', 'Alice',
↪  '2004-03-15', 'Info')")
cur.execute("INSERT INTO Etudiants VALUES (2, 'Petit', 'Bob',
↪  '2003-11-22', 'Info')")

conn.commit()

# Lecture
cur.execute("SELECT * FROM Etudiants")
for row in cur.fetchall():
    print(row)

conn.close()
```

Sortie

```
(1, 'Bernard', 'Alice', '2004-03-15', 'Info')
(2, 'Petit', 'Bob', '2003-11-22', 'Info')
```

Utiliser des requêtes paramétrées

Ne jamais construire des requêtes SQL par concaténation de chaînes : cela expose aux *injections SQL*. Utiliser les paramètres `?` :

```
# BON : requete parametree
cur.execute("SELECT * FROM Etudiants WHERE filiere = ?", ('Info',))

# MAUVAIS : concatenation (injection SQL possible !)
# cur.execute("SELECT * FROM Etudiants WHERE filiere = '" + filiere +
↪  "'")
```

1.8 Types de données SQL

Types de données courants

Catégorie	Type SQL	Description
Entier	`INTEGER`, `SMALLINT`, `BIGINT`	Nombres entiers
Réel	`REAL`, `FLOAT`, `DOUBLE`	Nombres à virgule flottante
Décimal	`NUMERIC(p,s)`, `DECIMAL`	Précision exacte
Texte	`TEXT`, `VARCHAR(n)`, `CHAR(n)`	Chaînes de caractères
Date/heure	`DATE`, `TIME`, `TIMESTAMP`	Temporel
Booléen	`BOOLEAN`	Vrai/faux
Binaire	`BLOB`	Données binaires

NULL n'est pas zéro

La valeur `NULL` représente l'absence de valeur. Elle est différente de 0, de la chaîne vide `''` ou de `FALSE`. Toute comparaison avec `NULL` retourne `UNKNOWN` (logique ternaire). On teste la nullité avec `IS NULL` ou `IS NOT NULL`, jamais avec `= NULL`.

1.9 Contraintes d'intégrité

Définition 1.11 (Contraintes d'intégrité)**.** Les contraintes d'intégrité sont des règles qui garantissent la cohérence des données dans la base. Les principales contraintes sont :

- `PRIMARY KEY` — identifie de manière unique chaque ligne.
- `FOREIGN KEY` — référence une clé primaire d'une autre table.
- `NOT NULL` — interdit les valeurs nulles.
- `UNIQUE` — interdit les doublons.
- `CHECK` — vérifie une condition logique.
- `DEFAULT` — valeur par défaut si aucune n'est fournie.

Exemple complet de contraintes

```
CREATE TABLE Inscriptions (
    id_etudiant INTEGER NOT NULL,
    id_cours    INTEGER NOT NULL,
    note        REAL CHECK (note >= 0 AND note <= 20),
    annee       INTEGER DEFAULT 2025,
    PRIMARY KEY (id_etudiant, id_cours),
    FOREIGN KEY (id_etudiant) REFERENCES Etudiants(id_etudiant)
        ON DELETE CASCADE,
    FOREIGN KEY (id_cours) REFERENCES Cours(id_cours)
        ON UPDATE CASCADE
);
```

1.10 Modèles de données : panorama

Outre le modèle relationnel, d'autres modèles existent :

Modèle	Structure	Exemple de SGBD
Relationnel	Tables (lignes/colonnes)	PostgreSQL, MySQL
Document	Documents JSON/BSON	MongoDB, CouchDB
Clé-valeur	Paires clé → valeur	Redis, DynamoDB
Colonne	Familles de colonnes	Cassandra, HBase
Graphe	Nœuds et arêtes	Neo4j, ArangoDB

Le chapitre 11 approfondira ces alternatives.

Exercices

Exercice 1.1. Donnez trois avantages d'un SGBD par rapport à un système de fichiers plat (CSV). Illustrez chaque avantage avec un exemple concret lié à la gestion d'une université.

Exercice 1.2. Pour chacune des tables suivantes, identifiez la ou les clés candidates :

1. `Livres(isbn, titre, auteur, annee_publication)`
2. `Employes(matricule, nom, prenom, email, departement)`
3. `Vols(compagnie, numero_vol, date_depart, aeroport_depart, aeroport_arrivee)`

Exercice 1.3. Écrivez le code SQL pour créer une base de données de bibliothèque avec les tables `Livres`, `Auteurs`, `Emprunts` et `Adherents`. Définissez les clés primaires, clés étrangères et au moins deux contraintes `CHECK`.

Exercice 1.4. Écrivez un script Python utilisant `sqlite3` qui :

1. Crée la base de données `bibliotheque.db`.
2. Crée les tables définies à l'exercice précédent.
3. Insère au moins 5 livres et 3 adhérents.
4. Affiche tous les livres dont l'année de publication est supérieure à 2020.

Exercice 1.5. Expliquez la différence entre indépendance logique et indépendance physique des données dans l'architecture ANSI/SPARC. Donnez un exemple concret pour chaque type.

Chapitre 2

Le Modèle Entité-Relation

Avant d'écrire la première ligne de SQL, il faut *penser* la structure des données. C'est l'étape de la *modélisation conceptuelle*, et l'outil le plus utilisé pour cela est le modèle entité-relation, proposé par Peter Chen en 1976. L'idée est de représenter le monde réel par des *entités* (les objets), des *attributs* (leurs propriétés) et des *relations* (les liens entre eux), dans un diagramme visuel qui sert de plan d'architecte pour la base de données.

2.1 Introduction à la conception de bases de données

La conception d'une base de données suit trois étapes principales :

1. **Analyse des besoins** — recueil des exigences auprès des utilisateurs.
2. **Conception conceptuelle** — modélisation indépendante de tout SGBD (modèle Entité-Relation).
3. **Conception logique** — traduction en schéma relationnel (tables SQL).
4. **Conception physique** — choix des index, du partitionnement, etc.

Le modèle Entité-Relation (E/R), introduit par Peter Chen en 1976, est l'outil standard pour la conception conceptuelle.

2.2 Concepts fondamentaux

Définition 2.1 (Entité)**.** Une *entité* est un objet du monde réel, distinguable des autres objets, sur lequel on souhaite stocker des informations. Un *type d'entité* regroupe les entités partageant les mêmes attributs.

Définition 2.2 (Attribut)**.** Un *attribut* est une propriété qui décrit une entité ou une relation. Chaque attribut a un *domaine* (ensemble des valeurs possibles). On distingue :

- **Attribut simple** vs. **composé** (ex. adresse = rue + ville + code postal).
- **Attribut mono-valué** vs. **multi-valué** (ex. téléphones).
- **Attribut stocké** vs. **dérivé** (ex. âge calculé à partir de la date de naissance).

Définition 2.3 (Clé d'entité)**.** Un attribut (ou ensemble d'attributs) dont la valeur identifie de manière unique chaque instance d'un type d'entité. On le souligne dans le diagramme E/R.

Définition 2.4 (Relation (association))**.** Une *relation* (ou *association*) représente un lien entre deux ou plusieurs types d'entités. Elle peut posséder des attributs propres.

2.3 Cardinalités

Définition 2.5 (Cardinalité)**.** La *cardinalité* d'une participation d'un type d'entité à une relation exprime le nombre minimal et maximal d'instances de la relation auxquelles une instance de l'entité peut participer. On note (min, max).

Les cardinalités les plus courantes sont :

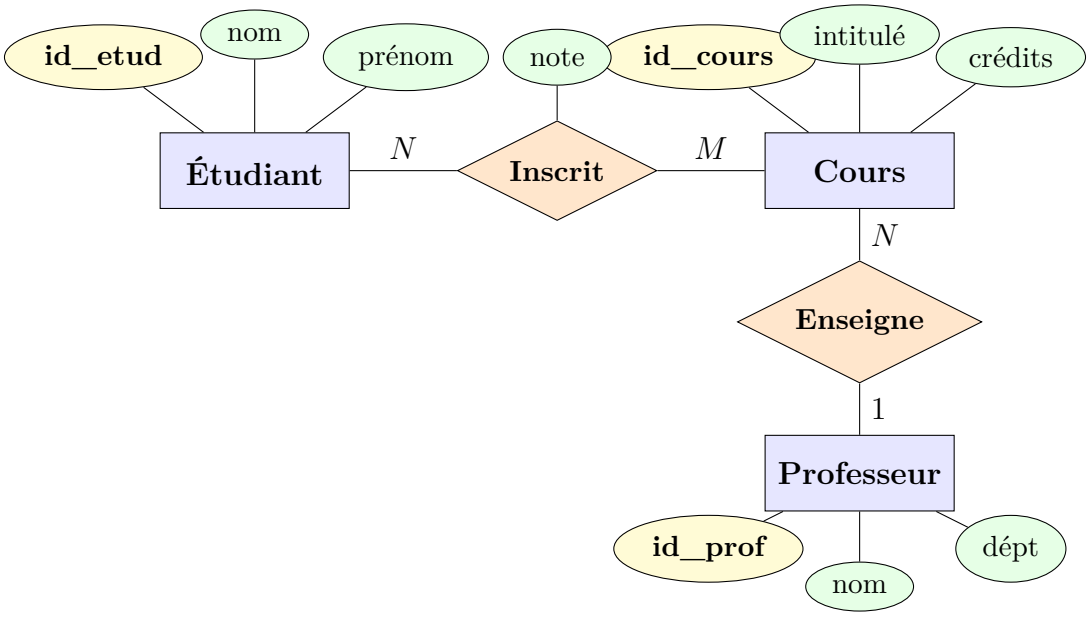
Types de cardinalités

Notation	Signification	Exemple
$1:1$ (un-à-un)	Chaque entité est liée à au plus une	Personne — Passeport
$1:N$ (un-à-plusieurs)	Une entité est liée à plusieurs	Professeur — Cours
$N:M$ (plusieurs-à-plusieurs)	Plusieurs à plusieurs	Étudiant — Cours

Exemple 2.6. Un professeur enseigne *plusieurs* cours, mais chaque cours est enseigné par *un seul* professeur : c'est une relation $1:N$.

Un étudiant peut s'inscrire à *plusieurs* cours, et chaque cours accueille *plusieurs* étudiants : c'est une relation $N:M$.

2.4 Diagrammes E/R avec TikZ



2.5 Entités faibles

Définition 2.7 (Entité faible)**.** Une *entité faible* est un type d'entité qui ne possède pas de clé propre suffisante pour l'identifier de manière unique. Son identification dépend d'un type d'entité *propriétaire* (entité forte) via une *relation identifiante*.

Exemple 2.8. Considérons les salles de cours d'un bâtiment. Une salle est identifiée par son numéro *dans* un bâtiment donné. Le type d'entité `Salle` est faible, identifié par la combinaison de `Batiment.id_bat` et `Salle.numero`.

2.6 Spécialisation et généralisation

Définition 2.9 (Spécialisation / Généralisation)**.** La *spécialisation* consiste à définir des sous-types d'un type d'entité général (relation « est-un », *is-a*). La *généralisation* est le processus inverse : regrouper des types d'entités partageant des attributs communs en un super-type.

Exemple 2.10. Le type `Personne` peut être spécialisé en `Étudiant` et `Professeur`. Les attributs communs (nom, prénom) sont dans `Personne`, les attributs spécifiques (filière pour Étudiant, département pour Professeur) sont dans les sous-types.

Contraintes de spécialisation :

- **Disjointe** (d) vs. **chevauchante** (o) : une entité peut-elle appartenir à plusieurs sous-types ?
- **Totale** (t) vs. **partielle** (p) : toute entité du super-type doit-elle appartenir à au moins un sous-type ?

2.7 Passage du modèle E/R au modèle relationnel

Théorème 2.11 (Règles de traduction E/R → Relationnel)**.** *Les règles de passage sont les suivantes :*

1. ***Type d'entité fort*** → *une table dont la clé primaire est la clé de l'entité.*
2. ***Type d'entité faible*** → *une table dont la clé primaire combine la clé partielle et la clé de l'entité propriétaire (clé étrangère).*
3. ***Relation*** $1 : N$ → *ajout d'une clé étrangère dans la table côté* N*.*
4. ***Relation*** $N : M$ → *création d'une table d'association dont la clé primaire est la paire des clés étrangères.*
5. ***Relation*** $1 : 1$ → *clé étrangère dans l'une des deux tables (de préférence celle avec participation totale).*

6. ***Attribut multi-valué*** → *table séparée avec clé étrangère.*

7. ***Spécialisation*** → *plusieurs stratégies possibles (tables séparées, table unique avec discriminateur, etc.).*

Exemple 2.12. Appliquons les règles au diagramme précédent :

Traduction du diagramme E/R en SQL

```sql
-- Regle 1 : entite forte -> table
CREATE TABLE Etudiants (
    id_etudiant INTEGER PRIMARY KEY,
    nom         TEXT NOT NULL,
    prenom      TEXT NOT NULL
);

CREATE TABLE Cours (
    id_cours    INTEGER PRIMARY KEY,
    intitule    TEXT NOT NULL,
    credits     INTEGER DEFAULT 3,
    id_prof     INTEGER REFERENCES Professeurs(id_prof)  -- Regle 3 : 1:N
);

CREATE TABLE Professeurs (
    id_prof     INTEGER PRIMARY KEY,
    nom         TEXT NOT NULL,
    departement TEXT
);

-- Regle 4 : relation N:M -> table d'association
CREATE TABLE Inscriptions (
    id_etudiant INTEGER REFERENCES Etudiants(id_etudiant),
    id_cours    INTEGER REFERENCES Cours(id_cours),
    note        REAL,       -- attribut de la relation
    PRIMARY KEY (id_etudiant, id_cours)
);
```

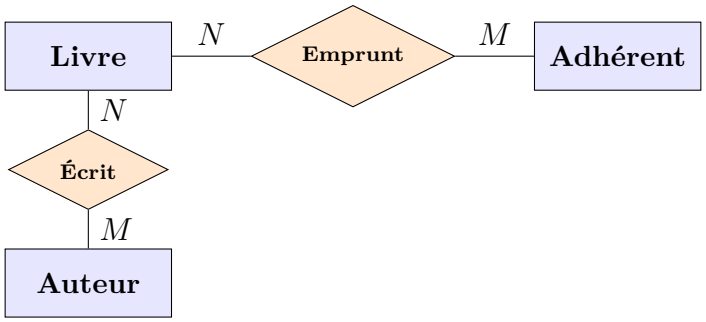
2.8 Exemple complet : système de bibliothèque

2.8.1 Analyse des besoins

Une bibliothèque souhaite gérer :

- Les **livres** (ISBN, titre, année de publication).
- Les **auteurs** (nom, nationalité). Un livre peut avoir plusieurs auteurs.
- Les **adhérents** (numéro, nom, adresse).
- Les **emprunts** (date d'emprunt, date de retour prévue, date de retour effective).

2.8.2 Diagramme E/R



2.8.3 Traduction en SQL

Schéma relationnel de la bibliothèque

```sql
CREATE TABLE Auteurs (
    id_auteur    INTEGER PRIMARY KEY,
    nom          TEXT NOT NULL,
    nationalite  TEXT
);

CREATE TABLE Livres (
    isbn         TEXT PRIMARY KEY,
    titre        TEXT NOT NULL,
    annee_pub    INTEGER
);

CREATE TABLE Ecrit_par (
    isbn         TEXT REFERENCES Livres(isbn),
    id_auteur    INTEGER REFERENCES Auteurs(id_auteur),
    PRIMARY KEY (isbn, id_auteur)
);

CREATE TABLE Adherents (
    id_adherent  INTEGER PRIMARY KEY,
    nom          TEXT NOT NULL,
    adresse      TEXT
);

CREATE TABLE Emprunts (
    id_emprunt   INTEGER PRIMARY KEY,
    isbn         TEXT REFERENCES Livres(isbn),
    id_adherent  INTEGER REFERENCES Adherents(id_adherent),
    date_emprunt DATE NOT NULL,
    date_retour_prevue DATE NOT NULL,
    date_retour_effective DATE
);
```

2.9 Bonnes pratiques de conception

Règles d'or de la conception E/R

1. Chaque type d'entité doit avoir une clé bien définie.
2. Éviter les attributs multi-valués : les transformer en entités séparées.
3. Privilégier les relations binaires aux relations ternaires (plus faciles à comprendre et à traduire).
4. Ne pas confondre entité et attribut : si un concept a ses propres attributs ou participe à des relations, c'est une entité.
5. Documenter chaque choix de conception (cardinalités, contraintes).

Anti-pattern : la table "fourre-tout"

Regrouper toutes les informations dans une seule table géante (étudiants, cours, notes, professeurs dans une même table) entraîne :

- Redondance massive des données.
- Anomalies d'insertion, de mise à jour et de suppression.
- Difficulté de maintenance et d'évolution.

La normalisation (chapitre 7) formalisera ces problèmes.

2.10 Intégration Python : génération de schéma

Génération automatique du schéma avec Python

```python
import sqlite3

schema = {
    'Auteurs': {
        'id_auteur': 'INTEGER PRIMARY KEY',
        'nom': 'TEXT NOT NULL',
        'nationalite': 'TEXT'
    },
    'Livres': {
        'isbn': 'TEXT PRIMARY KEY',
        'titre': 'TEXT NOT NULL',
        'annee_pub': 'INTEGER'
    }
}

conn = sqlite3.connect('bibliotheque.db')
cur = conn.cursor()
```

```
for table, colonnes in schema.items():
    cols = ', '.join(f'{col} {typ}' for col, typ in colonnes.items())
    sql = f'CREATE TABLE IF NOT EXISTS {table} ({cols})'
    print(f'Execution : {sql}')
    cur.execute(sql)

conn.commit()
conn.close()
```

Exercices

Exercice 2.1. Concevez un diagramme E/R pour un système de réservation d'hôtel comportant les entités `Client`, `Chambre`, `Hotel` et la relation `Reservation`. Précisez les cardinalités et les attributs de chaque entité et relation.

Exercice 2.2. Traduisez le diagramme E/R de l'exercice précédent en schéma relationnel SQL. Identifiez les clés primaires et étrangères.

Exercice 2.3. Un hôpital gère des `Patients`, des `Médecins`, des `Services` et des `Consultations`. Un patient peut consulter plusieurs médecins, un médecin appartient à un seul service.

1. Dessinez le diagramme E/R.
2. Identifiez une entité faible potentielle.
3. Traduisez en SQL.

Exercice 2.4. Quelle stratégie de traduction de spécialisation choisiriez-vous pour modéliser un système où `Véhicule` est spécialisé en `Voiture` et `Camion`, avec spécialisation disjointe et totale ? Justifiez votre choix et donnez le code SQL correspondant.

Exercice 2.5. Identifiez et corrigez les erreurs de conception dans le schéma suivant :

```
CREATE TABLE Commandes (
    id_commande INTEGER PRIMARY KEY,
    nom_client TEXT,
    adresse_client TEXT,
    telephone_client TEXT,
    produit TEXT,
    prix REAL,
    quantite INTEGER,
    nom_fournisseur TEXT,
    adresse_fournisseur TEXT
);
```

Proposez un diagramme E/R correct et le schéma SQL correspondant.

Chapitre 3

Algèbre Relationnelle

3.1 Introduction

Quand vous écrivez `SELECT nom FROM etudiants WHERE note > 15`, vous pensez en SQL. Mais derrière cette syntaxe se cache un langage mathématique d'une précision remarquable : l'*algèbre relationnelle*, inventée par Edgar F. Codd dans son article fondateur de 1970. Codd a eu l'intuition géniale de voir les tables comme des *relations* au sens mathématique (des sous-ensembles de produits cartésiens) et les requêtes comme des *opérations* sur ces relations : sélection, projection, jointure, union, différence. L'algèbre relationnelle est à SQL ce que la logique formelle est au langage naturel : un socle rigoureux qui garantit que chaque requête a une sémantique précise et non ambiguë.

Comprendre l'algèbre relationnelle, c'est comprendre *pourquoi* SQL fonctionne — et comment l'optimiseur de requêtes transforme vos instructions en un plan d'exécution efficace.

Définition 3.1 (Algèbre relationnelle)**.** L'*algèbre relationnelle* est un ensemble d'opérateurs qui prennent une ou deux relations en entrée et produisent une relation en sortie. C'est un langage *procédural* : on spécifie *comment* obtenir le résultat (séquence d'opérations).

Les opérateurs se divisent en deux catégories :

- **Opérateurs ensemblistes** : union ($\cup$), intersection ($\cap$), différence ($-$), produit cartésien ($\times$).
- **Opérateurs spécifiques** : sélection (σ), projection (π), jointure ($\bowtie$), renommage (ρ), division ($\div$).

3.2 Sélection (σ)

Définition 3.2 (Sélection)**.** La *sélection* $\sigma_\theta(R)$ retourne les tuples de R qui satisfont la condition θ. La condition θ est une expression booléenne portant sur les attributs de R.

$$\sigma_\theta(R) = \{t \in R \mid \theta(t) = \text{vrai}\}$$

Propriétés de la sélection

$$\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R)) \quad \text{(cascade)}$$
$$\sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R)) \quad \text{(commutativité)}$$
$$|\sigma_{\theta}(R)| \leq |R| \quad \text{(cardinalité)}$$

Exemple 3.3. Sélectionner les étudiants de la filière Informatique :

$$\sigma_{\text{filiere='Info'}}(\text{Etudiants})$$

Cela correspond en SQL à :

Équivalent SQL de la sélection

```sql
SELECT * FROM Etudiants WHERE filiere = 'Info';
```

Sortie

```
id_etudiant | nom     | prenom | date_naissance | filiere
------------+---------+--------+----------------+--------
1           | Bernard | Alice  | 2004-03-15     | Info
2           | Petit   | Bob    | 2003-11-22     | Info
4           | Leroy   | David  | 2003-01-30     | Info
```

3.3 Projection (π)

Définition 3.4 (Projection)**.** La *projection* $\pi_{A_1,A_2,\ldots,A_k}(R)$ retourne la relation obtenue en ne conservant que les attributs $A_1, \ldots, A_k$ de R, et en éliminant les doublons.

$$\pi_{A_1,\ldots,A_k}(R) = \{t[A_1, \ldots, A_k] \mid t \in R\}$$

Exemple 3.5. Projeter les noms et prénoms des étudiants :

$$\pi_{\text{nom,prenom}}(\text{Etudiants})$$

Équivalent SQL de la projection

```sql
SELECT DISTINCT nom, prenom FROM Etudiants;
```

Projection et doublons

En algèbre relationnelle, la projection élimine automatiquement les doublons (car une relation est un ensemble). En SQL, il faut utiliser `DISTINCT` explicitement, car SQL travaille avec des *multi-ensembles* (bags).

3.4 Opérateurs ensemblistes

Définition 3.6 (Union, Intersection, Différence)**.** Soient R et S deux relations de même schéma (*union-compatibles*) :

$$R \cup S = \{t \mid t \in R \vee t \in S\} \quad \text{(union)}$$
$$R \cap S = \{t \mid t \in R \wedge t \in S\} \quad \text{(intersection)}$$
$$R - S = \{t \mid t \in R \wedge t \notin S\} \quad \text{(différence)}$$

Théorème 3.7 (Propriétés ensemblistes)**.**

- $\cup$ *et* $\cap$ *sont commutatives et associatives.*
- $-$ *n'est ni commutative ni associative.*
- $R \cap S = R - (R - S)$.

Opérateurs ensemblistes en SQL

```sql
-- Union (elimine les doublons)
SELECT nom FROM Etudiants WHERE filiere = 'Info'
UNION
SELECT nom FROM Etudiants WHERE filiere = 'Maths';

-- Intersection
SELECT id_etudiant FROM Inscriptions WHERE id_cours = 101
INTERSECT
SELECT id_etudiant FROM Inscriptions WHERE id_cours = 102;

-- Difference
SELECT id_etudiant FROM Inscriptions WHERE id_cours = 101
EXCEPT
SELECT id_etudiant FROM Inscriptions WHERE id_cours = 103;
```

3.5 Produit cartésien ($\times$)

Définition 3.8 (Produit cartésien)**.** Le *produit cartésien* $R \times S$ produit toutes les combinaisons possibles de tuples de R avec des tuples de S.

$$R \times S = \{(t_r, t_s) \mid t_r \in R \wedge t_s \in S\}$$

Si $|R| = n$ et $|S| = m$, alors $|R \times S| = n \times m$.

Coût du produit cartésien

Le produit cartésien est très coûteux : il multiplie les cardinalités. En pratique, on l'utilise rarement seul ; on le combine avec une sélection pour obtenir une *jointure*.

3.6 Jointure ($\bowtie$)

Définition 3.9 (Jointure naturelle)**.** La *jointure naturelle* $R \bowtie S$ combine les tuples de R et S qui ont les mêmes valeurs sur les attributs communs.

$$R \bowtie S = \sigma_{\text{attributs communs égaux}}(R \times S)$$

avec élimination des colonnes dupliquées.

Définition 3.10 (Theta-jointure)**.** La *theta-jointure* $R \bowtie_\theta S$ est définie par :

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

où θ est une condition quelconque portant sur les attributs de R et S. Cas particulier : l'*équi-jointure* utilise uniquement des égalités.

Exemple 3.11. Trouver les noms des étudiants et les intitulés des cours auxquels ils sont inscrits :

$$\pi_{\text{nom,intitule}}(\text{Etudiants} \bowtie \text{Inscriptions} \bowtie \text{Cours})$$

Jointure en SQL

```sql
SELECT e.nom, c.intitule
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
JOIN Cours c ON i.id_cours = c.id_cours;
```

Sortie

```
nom     | intitule
--------+------------------
Bernard | Bases de donnees
Bernard | Algorithmes
Petit   | Bases de donnees
Petit   | Reseaux
Moreau  | Algorithmes
Moreau  | Statistiques
Leroy   | Bases de donnees
Roux    | Statistiques
```

Proposition 3.12 (Propriétés de la jointure)**.**

- La jointure naturelle est commutative : $R \bowtie S = S \bowtie R$.
- La jointure naturelle est associative : $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.
- Si R et S n'ont aucun attribut commun, $R \bowtie S = R \times S$.

3.7 Renommage (ρ)

Définition 3.13 (Renommage)**.** L'opérateur de *renommage* $\rho_{B/A}(R)$ renomme l'attribut A en B dans la relation R. On peut aussi renommer la relation elle-même : $\rho_S(R)$.

Le renommage est utile pour :

- Rendre deux relations union-compatibles.
- Réaliser une auto-jointure (jointure d'une relation avec elle-même).

Exemple 3.14. Trouver les paires d'étudiants inscrits au même cours :

$$\pi_{i_1.\text{id\_etud},i_2.\text{id\_etud}}(\sigma_{i_1.\text{id\_etud}<i_2.\text{id\_etud}}(\rho_{i_1}(\text{Inscriptions}) \bowtie_{\text{id\_cours}} \rho_{i_2}(\text{Inscriptions})))$$

3.8 Division (÷)

Définition 3.15 (Division)**.** Soient $R(A, B)$ et $S(B)$ deux relations. La *division* $R \div S$ retourne les valeurs de A dans R qui sont associées à *toutes* les valeurs de B dans S.

$$R \div S = \{t[A] \mid t \in R \wedge \forall s \in S, (t[A], s[B]) \in R\}$$

Théorème 3.16 (Expression de la division)**.** *La division peut s'exprimer en termes des autres opérateurs :*

$$R \div S = \pi_A(R) - \pi_A\big((\pi_A(R) \times S) - R\big)$$

Exemple 3.17. Trouver les étudiants inscrits à *tous* les cours du professeur 2 :
Soit $S = \pi_{\text{id\_cours}}(\sigma_{\text{id\_prof}=2}(\text{Cours}))$ (les cours du professeur 2).
$\pi_{\text{id\_etudiant},\text{id\_cours}}(\text{Inscriptions}) \div S$
donne les étudiants inscrits à tous ces cours.

Division en SQL (NOT EXISTS)

```sql
SELECT DISTINCT i.id_etudiant
FROM Inscriptions i
WHERE NOT EXISTS (
    SELECT c.id_cours
    FROM Cours c
    WHERE c.id_prof = 2
    AND NOT EXISTS (
        SELECT 1
        FROM Inscriptions i2
        WHERE i2.id_etudiant = i.id_etudiant
        AND i2.id_cours = c.id_cours
    )
);
```

3.9 Arbre algébrique et optimisation

Une expression d'algèbre relationnelle peut être représentée sous forme d'arbre : les feuilles sont les relations de base, les nœuds internes sont les opérateurs.

Théorème 3.18 (Règles d'optimisation heuristique)**.** *1.* ***Descendre les sélections*** *: appliquer les sélections le plus tôt possible pour réduire la taille des relations intermédiaires.*

2. ***Descendre les projections*** *: projeter le plus tôt possible pour réduire le nombre d'attributs.*

3. ***Regrouper sélection + produit cartésien en jointure*** *:* $\sigma_\theta(R \times S) \to R \bowtie_\theta S$.

4. ***Réordonner les jointures*** *: commencer par les jointures qui produisent les plus petits résultats intermédiaires.*

Exemple 3.19. Considérons la requête : « noms des étudiants en Info inscrits au cours 101 ».

Expression naïve :

$$\pi_{\text{nom}}(\sigma_{\text{filiere='Info'} \wedge \text{id\_cours}=101}(\text{Etudiants} \times \text{Inscriptions}))$$

Expression optimisée :

$$\pi_{\text{nom}}(\sigma_{\text{filiere='Info'}}(\text{Etudiants}) \bowtie \sigma_{\text{id\_cours}=101}(\text{Inscriptions}))$$

La version optimisée évite le produit cartésien complet et filtre au plus tôt.

3.10 Résumé des correspondances algèbre–SQL

Correspondance Algèbre Relationnelle ↔ SQL

Algèbre	SQL
$\sigma_\theta(R)$	`SELECT * FROM R WHERE` θ
$\pi_{A_1,\ldots,A_k}(R)$	`SELECT DISTINCT` A_1, …, A_k `FROM R`
$R \cup S$	`R UNION S`
$R \cap S$	`R INTERSECT S`
$R - S$	`R EXCEPT S`
$R \times S$	`SELECT * FROM R, S`
$R \bowtie S$	`SELECT * FROM R NATURAL JOIN S`
$R \bowtie_\theta S$	`SELECT * FROM R JOIN S ON` θ
$R \div S$	Double `NOT EXISTS`
$\rho_{B/A}(R)$	`SELECT A AS B FROM R`

3.11 Intégration Python

Simulation de l'algèbre relationnelle avec pandas

```python
import pandas as pd

# Relations comme DataFrames
etudiants = pd.DataFrame({
    'id_etudiant': [1, 2, 3, 4, 5],
    'nom': ['Bernard', 'Petit', 'Moreau', 'Leroy', 'Roux'],
    'filiere': ['Info', 'Info', 'Maths', 'Info', 'Maths']
```

```
})
inscriptions = pd.DataFrame({
    'id_etudiant': [1, 1, 2, 2, 3, 3, 4, 5],
    'id_cours': [101, 102, 101, 103, 102, 104, 101, 104],
    'note': [15.5, 13.0, 12.0, 16.0, 17.5, 14.0, 9.0, 18.0]
})

# Selection : sigma_{filiere='Info'}(Etudiants)
info = etudiants[etudiants['filiere'] == 'Info']
print("Selection :\n", info)

# Projection : pi_{nom}(Etudiants)
noms = etudiants[['nom']].drop_duplicates()
print("\nProjection :\n", noms)

# Jointure naturelle
resultat = pd.merge(etudiants, inscriptions, on='id_etudiant')
print("\nJointure :\n", resultat[['nom', 'id_cours', 'note']])
```

Exercices

Exercice 3.1. Exprimez en algèbre relationnelle les requêtes suivantes, puis traduisez-les en SQL :

1. Les noms des professeurs du département “Informatique”.
2. Les intitulés des cours ayant au moins 4 crédits.
3. Les noms des étudiants inscrits au cours “Bases de données”.

Exercice 3.2. Soit $R(A, B, C)$ et $S(C, D, E)$. Simplifiez l'expression :

$$\pi_{A,D}(\sigma_{B>10 \wedge D='x'}(R \bowtie S))$$

en descendant les sélections et projections.

Exercice 3.3. Exprimez la division suivante en SQL : « trouver les étudiants inscrits à tous les cours du département Informatique ».

Exercice 3.4. Dessinez l'arbre algébrique correspondant à l'expression :

$$\pi_{\text{nom}}(\sigma_{\text{note}>15}(\text{Etudiants} \bowtie \text{Inscriptions} \bowtie \text{Cours}))$$

Puis proposez un arbre optimisé en appliquant les règles heuristiques.

Exercice 3.5. Prouvez que $R \cap S = R - (R - S)$ en utilisant la définition ensembliste des opérateurs.

Chapitre 4

SQL — Requêtes de Base

En 1974, Donald Chamberlin et Raymond Boyce, chercheurs chez IBM, créent SEQUEL (*Structured English Query Language*), rebaptisé SQL pour des raisons juridiques. Leur ambition : permettre à des non-programmeurs d'interroger des bases de données en écrivant des phrases proches de l'anglais. L'idée était révolutionnaire : au lieu de dire *comment* parcourir les données, on dit simplement *ce que* l'on veut. Cinquante ans plus tard, SQL reste le langage universel des bases de données relationnelles.

4.1 Introduction à SQL

Définition 4.1 (SQL)**.** *SQL* (*Structured Query Language*) est le langage standard pour interroger et manipuler les bases de données relationnelles. Contrairement à l'algèbre relationnelle, SQL est un langage *déclaratif* : on spécifie *ce que* l'on veut, pas *comment* l'obtenir.

SQL a été standardisé par l'ISO/ANSI. Les versions majeures sont : SQL-86, SQL-92, SQL :1999, SQL :2003, SQL :2011, SQL :2016, SQL :2023.

4.2 Structure d'une requête SELECT

Syntaxe générale du SELECT

```
SELECT [DISTINCT] colonnes
FROM   tables
[WHERE  condition]
[ORDER BY colonne [ASC|DESC]]
[LIMIT  n [OFFSET m]];
```

L'ordre d'exécution logique est différent de l'ordre d'écriture :

1. `FROM` — déterminer les tables sources.
2. `WHERE` — filtrer les lignes.
3. `SELECT` — projeter les colonnes.
4. `DISTINCT` — éliminer les doublons.

5. `ORDER BY` — trier le résultat.

6. `LIMIT / OFFSET` — limiter le nombre de lignes.

4.3 SELECT et FROM

Sélectionner toutes les colonnes

```sql
SELECT * FROM Etudiants;
```

Sortie

```
id_etudiant | nom     | prenom | date_naissance | filiere
------------+---------+--------+----------------+--------
1           | Bernard | Alice  | 2004-03-15     | Info
2           | Petit   | Bob    | 2003-11-22     | Info
3           | Moreau  | Claire | 2004-07-08     | Maths
4           | Leroy   | David  | 2003-01-30     | Info
5           | Roux    | Emma   | 2004-09-12     | Maths
```

Sélectionner des colonnes spécifiques

```sql
SELECT nom, prenom, filiere FROM Etudiants;
```

Éviter SELECT *

En production, évitez `SELECT *` :

- Cela transfère des données inutiles.
- Le résultat dépend du schéma : l'ajout d'une colonne peut casser le code.
- Utilisez `SELECT *` uniquement pour l'exploration et le débogage.

4.4 Alias de colonnes et de tables

Utilisation d'alias

```sql
SELECT e.nom AS nom_etudiant,
       e.prenom AS prenom_etudiant,
       e.filiere AS departement
FROM Etudiants AS e;
```

Le mot-clé `AS` est optionnel dans la plupart des SGBD : `SELECT nom nom_etudiant FROM Etudiants e`.

4.5 Clause WHERE — Filtrage

Définition 4.2 (Clause WHERE)**.** La clause `WHERE` filtre les lignes selon une condition booléenne. Seules les lignes pour lesquelles la condition est `TRUE` sont retenues (`FALSE` et `UNKNOWN` sont rejetés).

4.5.1 Opérateurs de comparaison

Opérateurs de comparaison SQL

Opérateur	Signification
`=`	Égal
`<>` ou `!=`	Différent
`<`, `>`	Inférieur, supérieur
`<=`, `>=`	Inférieur ou égal, supérieur ou égal
`BETWEEN a AND b`	Entre a et b (inclus)
`IN (v1, v2, ...)`	Appartient à une liste
`LIKE 'motif'`	Correspondance de motif
`IS NULL` / `IS NOT NULL`	Test de nullité

Exemples de filtrage

```sql
-- Etudiants en Info
SELECT * FROM Etudiants WHERE filiere = 'Info';

-- Notes entre 10 et 15
SELECT * FROM Inscriptions WHERE note BETWEEN 10 AND 15;

-- Etudiants dont le nom commence par 'B'
SELECT * FROM Etudiants WHERE nom LIKE 'B%';

-- Etudiants en Info ou Maths
SELECT * FROM Etudiants WHERE filiere IN ('Info', 'Maths');

-- Inscriptions sans note
SELECT * FROM Inscriptions WHERE note IS NULL;
```

4.5.2 Opérateurs logiques

Combinaison de conditions

```sql
-- AND : les deux conditions doivent etre vraies
SELECT * FROM Etudiants
WHERE filiere = 'Info' AND date_naissance > '2004-01-01';

-- OR : au moins une condition vraie
SELECT * FROM Inscriptions
```

```sql
WHERE note > 16 OR note < 8;

-- NOT : negation
SELECT * FROM Etudiants
WHERE NOT filiere = 'Maths';
```

Logique ternaire et NULL

En SQL, la logique est *ternaire* : `TRUE`, `FALSE`, `UNKNOWN`.

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

`NOT UNKNOWN = UNKNOWN`. Conséquence : `note = NULL` retourne `UNKNOWN`, pas `TRUE`. Utilisez `IS NULL`.

4.5.3 LIKE et les motifs

Caractères spéciaux de LIKE

Caractère	Signification
%	Zéro ou plusieurs caractères quelconques
_	Exactement un caractère quelconque

Exemples avec LIKE

```sql
-- Noms commencant par 'M'
SELECT * FROM Etudiants WHERE nom LIKE 'M%';

-- Prenoms de exactement 5 lettres
SELECT * FROM Etudiants WHERE prenom LIKE '_____';

-- Noms contenant 'or'
SELECT * FROM Etudiants WHERE nom LIKE '%or%';
```

4.6 ORDER BY — Tri

Tri des résultats

```sql
-- Tri ascendant (par defaut)
SELECT nom, prenom FROM Etudiants ORDER BY nom ASC;

-- Tri descendant
SELECT * FROM Inscriptions ORDER BY note DESC;
```

```sql
-- Tri multi-colonnes
SELECT * FROM Etudiants ORDER BY filiere ASC, nom ASC;
```

Sortie

```
-- Tri par note descendante :
id_etudiant | id_cours | note | annee
-----------+---------+------+------
5           | 104      | 18.0 | 2025
3           | 102      | 17.5 | 2025
2           | 103      | 16.0 | 2025
1           | 101      | 15.5 | 2025
3           | 104      | 14.0 | 2025
1           | 102      | 13.0 | 2025
2           | 101      | 12.0 | 2025
4           | 101      | 9.0  | 2025
```

4.7 DISTINCT — Élimination des doublons

Utilisation de DISTINCT

```sql
-- Sans DISTINCT : doublons possibles
SELECT filiere FROM Etudiants;
-- Info, Info, Maths, Info, Maths

-- Avec DISTINCT : valeurs uniques
SELECT DISTINCT filiere FROM Etudiants;
-- Info, Maths
```

4.8 LIMIT et OFFSET — Pagination

Limiter les résultats

```sql
-- Les 3 meilleures notes
SELECT e.nom, i.note
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
ORDER BY i.note DESC
LIMIT 3;

-- Pagination : page 2 (lignes 4 a 6)
SELECT * FROM Etudiants ORDER BY id_etudiant LIMIT 3 OFFSET 3;
```

Sortie

```
-- Les 3 meilleures notes :
nom    | note
-------+-----
Roux   | 18.0
Moreau | 17.5
Petit  | 16.0
```

LIMIT n'est pas standard SQL

`LIMIT` est supporté par PostgreSQL, MySQL, SQLite. Le standard SQL :2008 utilise `FETCH FIRST n ROWS ONLY` :

```sql
SELECT * FROM Etudiants
ORDER BY nom
FETCH FIRST 3 ROWS ONLY;  -- Standard SQL
```

4.9 Expressions et fonctions scalaires

Expressions calculées

```sql
-- Expression arithmetique
SELECT nom, note, note / 20.0 * 100 AS pourcentage
FROM Inscriptions i
JOIN Etudiants e ON i.id_etudiant = e.id_etudiant;

-- Concatenation de chaines
SELECT nom || ' ' || prenom AS nom_complet
FROM Etudiants;

-- Fonction conditionnelle CASE
SELECT nom, note,
    CASE
        WHEN note >= 16 THEN 'Tres bien'
        WHEN note >= 14 THEN 'Bien'
        WHEN note >= 12 THEN 'Assez bien'
        WHEN note >= 10 THEN 'Passable'
        ELSE 'Insuffisant'
    END AS mention
FROM Inscriptions i
JOIN Etudiants e ON i.id_etudiant = e.id_etudiant;
```

Sortie

```
nom      | note | mention
---------+------+-----------
```

```
Bernard | 15.5 | Bien
Bernard | 13.0 | Assez bien
Petit   | 12.0 | Assez bien
Petit   | 16.0 | Tres bien
Moreau  | 17.5 | Tres bien
Moreau  | 14.0 | Bien
Leroy   | 9.0  | Insuffisant
Roux    | 18.0 | Tres bien
```

4.10 Fonctions utiles

Fonctions scalaires courantes

Catégorie	Fonction	Description
Texte	`UPPER(s)`, `LOWER(s)`	Majuscules, minuscules
	`LENGTH(s)`	Longueur
	`SUBSTR(s, d, n)`	Sous-chaîne
	`TRIM(s)`	Suppression espaces
	`REPLACE(s, a, b)`	Remplacement
Nombre	`ABS(x)`, `ROUND(x, n)`	Valeur absolue, arrondi
	`CEIL(x)`, `FLOOR(x)`	Plafond, plancher
Date	`CURRENT_DATE`	Date courante
	`DATE('now')`	Date courante (SQLite)
NULL	`COALESCE(a, b, c)`	Première valeur non NULL
	`NULLIF(a, b)`	NULL si $a = b$

Exemples de fonctions

```sql
SELECT UPPER(nom) AS nom_maj,
       LENGTH(prenom) AS lg_prenom,
       COALESCE(note, 0) AS note_ou_zero
FROM Etudiants e
LEFT JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant;
```

4.11 INSERT, UPDATE, DELETE

Insertion de données

```sql
-- Insertion d'une ligne
INSERT INTO Etudiants (id_etudiant, nom, prenom, filiere)
VALUES (6, 'Garcia', 'Lucie', 'Info');

-- Insertion multiple
INSERT INTO Etudiants (id_etudiant, nom, prenom, filiere) VALUES
```

```sql
    (7, 'Dubois', 'Marc', 'Maths'),
    (8, 'Thomas', 'Julie', 'Info');

-- Insertion depuis une requete
INSERT INTO Archive_Etudiants
SELECT * FROM Etudiants WHERE filiere = 'Maths';
```

Mise à jour et suppression

```sql
-- Mise a jour
UPDATE Inscriptions SET note = 11.0
WHERE id_etudiant = 4 AND id_cours = 101;

-- Suppression conditionnelle
DELETE FROM Inscriptions WHERE note < 8;

-- Suppression de toutes les lignes
DELETE FROM Archive_Etudiants;
-- ou TRUNCATE TABLE Archive_Etudiants;  (PostgreSQL)
```

UPDATE/DELETE sans WHERE

Un `UPDATE` ou `DELETE` sans clause `WHERE` affecte *toutes* les lignes de la table. Toujours vérifier la clause `WHERE` avant d'exécuter !

4.12 Intégration Python complète

CRUD complet avec Python/sqlite3

```python
import sqlite3

conn = sqlite3.connect(':memory:')  # base en memoire
cur = conn.cursor()

# Creation et insertion
cur.executescript('''
    CREATE TABLE Etudiants (
        id INTEGER PRIMARY KEY, nom TEXT, prenom TEXT, filiere TEXT
    );
    INSERT INTO Etudiants VALUES (1, 'Bernard', 'Alice', 'Info');
    INSERT INTO Etudiants VALUES (2, 'Petit', 'Bob', 'Info');
    INSERT INTO Etudiants VALUES (3, 'Moreau', 'Claire', 'Maths');
''')

# Lecture avec parametres
filiere = 'Info'
cur.execute("SELECT * FROM Etudiants WHERE filiere = ?", (filiere,))
```

```python
print("Etudiants Info :", cur.fetchall())

# Mise a jour
cur.execute("UPDATE Etudiants SET filiere = ? WHERE id = ?", ('Maths',
↪  2))
conn.commit()

# Verification
cur.execute("SELECT * FROM Etudiants ORDER BY id")
for row in cur.fetchall():
    print(row)

conn.close()
```

Sortie

```
Etudiants Info : [(1, 'Bernard', 'Alice', 'Info'), (2, 'Petit', 'Bob', 'Info')]
(1, 'Bernard', 'Alice', 'Info')
(2, 'Petit', 'Bob', 'Maths')
(3, 'Moreau', 'Claire', 'Maths')
```

Exercices

Exercice 4.1. Écrivez les requêtes SQL suivantes :

1. Tous les cours ayant plus de 3 crédits.
2. Les noms et prénoms des étudiants nés avant 2004.
3. Les inscriptions avec une note supérieure à 15, triées par note décroissante.
4. Les 3 étudiants ayant les meilleures notes.

Exercice 4.2. Expliquez la différence entre les deux requêtes :

```sql
SELECT DISTINCT filiere FROM Etudiants;
SELECT filiere FROM Etudiants GROUP BY filiere;
```

Exercice 4.3. Écrivez une requête SQL qui affiche pour chaque inscription : le nom de l'étudiant, l'intitulé du cours, la note, et la mention correspondante (Très bien ≥ 16, Bien ≥ 14, Assez bien ≥ 12, Passable ≥ 10, Insuffisant sinon).

Exercice 4.4. Quels sont les résultats des expressions suivantes ?

1. `NULL = NULL`
2. `NULL <> 5`
3. `NULL AND TRUE`

4. `NULL OR TRUE`
5. `NOT NULL`

Exercice 4.5. Écrivez un programme Python qui :

1. Crée une base de données SQLite avec les tables du schéma Université.
2. Insère les données d'exemple.
3. Exécute 3 requêtes différentes et affiche les résultats formatés.
4. Utilise des requêtes paramétrées pour toutes les entrées utilisateur.

Chapitre 5

SQL — Jointures et Sous-requêtes

La puissance du modèle relationnel réside dans sa capacité à relier des tables entre elles. Un étudiant est inscrit dans un département, qui appartient à une université ; chaque inscription référence un cours, qui est enseigné par un professeur. Pour extraire l'information complète — par exemple, « quels étudiants suivent les cours du Professeur Dupont ? »— il faut *joindre* plusieurs tables. La jointure est l'opération reine de SQL, et sa maîtrise distingue le débutant de l'expert. Les sous-requêtes, quant à elles, permettent d'imbriquer des questions les unes dans les autres, offrant une expressivité considérable.

5.1 Introduction

Les *jointures* permettent de combiner des données provenant de plusieurs tables en exploitant les relations entre elles (clés étrangères). Les *sous-requêtes* (ou requêtes imbriquées) permettent d'utiliser le résultat d'une requête comme entrée d'une autre.

5.2 Jointure interne (INNER JOIN)

Définition 5.1 (Jointure interne)**.** La *jointure interne* combine les lignes de deux tables où la condition de jointure est satisfaite. Les lignes sans correspondance sont éliminées.

Syntaxes équivalentes de la jointure interne

```sql
-- Syntaxe explicite (recommandee)
SELECT e.nom, c.intitule, i.note
FROM Etudiants e
INNER JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
INNER JOIN Cours c ON i.id_cours = c.id_cours;

-- Syntaxe implicite (ancienne, a eviter)
SELECT e.nom, c.intitule, i.note
FROM Etudiants e, Inscriptions i, Cours c
WHERE e.id_etudiant = i.id_etudiant
  AND i.id_cours = c.id_cours;
```

Sortie

```
nom     | intitule          | note
--------+-------------------+------
Bernard | Bases de donnees  | 15.5
Bernard | Algorithmes       | 13.0
Petit   | Bases de donnees  | 12.0
Petit   | Reseaux           | 16.0
Moreau  | Algorithmes       | 17.5
Moreau  | Statistiques      | 14.0
Leroy   | Bases de donnees  | 9.0
Roux    | Statistiques      | 18.0
```

Toujours utiliser la syntaxe explicite JOIN

La syntaxe implicite (virgule dans le `FROM`) est source d'erreurs : un oubli de condition dans le `WHERE` produit un produit cartésien accidentel. La syntaxe `JOIN ... ON` est plus claire et séparere la logique de jointure du filtrage.

5.3 Jointure naturelle (NATURAL JOIN)

Jointure naturelle

```sql
-- Joint sur les colonnes de meme nom
SELECT * FROM Etudiants NATURAL JOIN Inscriptions;
```

Danger de NATURAL JOIN

La jointure naturelle détermine automatiquement les colonnes communes. Si deux tables partagent un nom de colonne par coïncidence (ex. `nom` dans `Etudiants` et `Professeurs`), le résultat sera incorrect. Préférez toujours la jointure explicite avec `ON`.

5.4 Jointures externes (OUTER JOIN)

Définition 5.2 (Jointure externe)**.** Une *jointure externe* conserve toutes les lignes d'une table (ou des deux), même celles sans correspondance dans l'autre table. Les colonnes manquantes sont remplies par `NULL`.

5.4.1 LEFT JOIN

LEFT JOIN — tous les étudiants, même sans inscription

```sql
SELECT e.nom, e.prenom, c.intitule, i.note
FROM Etudiants e
LEFT JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
LEFT JOIN Cours c ON i.id_cours = c.id_cours
ORDER BY e.nom;
```

Si un étudiant n'est inscrit à aucun cours, il apparaît quand même avec des `NULL` pour `intitule` et `note`.

5.4.2 RIGHT JOIN

RIGHT JOIN — tous les cours, même sans inscription

```sql
SELECT c.intitule, e.nom, i.note
FROM Inscriptions i
RIGHT JOIN Cours c ON i.id_cours = c.id_cours
LEFT JOIN Etudiants e ON i.id_etudiant = e.id_etudiant;
```

Remarque 5.3. `RIGHT JOIN` est moins utilisé car on peut toujours le remplacer par un `LEFT JOIN` en inversant l'ordre des tables. SQLite ne supporte pas `RIGHT JOIN` (avant la version 3.39).

5.4.3 FULL OUTER JOIN

FULL OUTER JOIN

```sql
-- Tous les etudiants et tous les cours (PostgreSQL)
SELECT e.nom, c.intitule
FROM Etudiants e
FULL OUTER JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
FULL OUTER JOIN Cours c ON i.id_cours = c.id_cours;
```

Résumé des types de jointures

Type	Description
`INNER JOIN`	Lignes avec correspondance dans les deux tables
`LEFT JOIN`	Toutes les lignes de la table gauche + correspondances
`RIGHT JOIN`	Toutes les lignes de la table droite + correspondances
`FULL OUTER JOIN`	Toutes les lignes des deux tables
`CROSS JOIN`	Produit cartésien (toutes les combinaisons)

5.5 Jointure croisée (CROSS JOIN)

Produit cartésien

```sql
-- Toutes les combinaisons etudiant-cours
SELECT e.nom, c.intitule
FROM Etudiants e
CROSS JOIN Cours c;
-- 5 etudiants x 4 cours = 20 lignes
```

5.6 Auto-jointure (Self Join)

Définition 5.4 (Auto-jointure)**.** Une *auto-jointure* est une jointure d'une table avec elle-même. Elle nécessite l'utilisation d'alias pour distinguer les deux « copies »de la table.

Trouver les paires d'étudiants de la même filière

```sql
SELECT e1.nom AS etudiant1, e2.nom AS etudiant2, e1.filiere
FROM Etudiants e1
JOIN Etudiants e2 ON e1.filiere = e2.filiere
                  AND e1.id_etudiant < e2.id_etudiant;
```

Sortie

```
etudiant1 | etudiant2 | filiere
----------+-----------+--------
Bernard   | Petit     | Info
Bernard   | Leroy     | Info
Petit     | Leroy     | Info
Moreau    | Roux      | Maths
```

5.7 Sous-requêtes

Définition 5.5 (Sous-requête)**.** Une *sous-requête* est une requête `SELECT` imbriquée dans une autre requête. Elle peut apparaître dans les clauses `WHERE`, `FROM`, ou `SELECT`.

5.7.1 Sous-requête scalaire

Une sous-requête qui retourne exactement une valeur :

Sous-requête scalaire

```sql
-- Etudiants ayant une note superieure a la moyenne
SELECT e.nom, i.note
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
WHERE i.note > (SELECT AVG(note) FROM Inscriptions);
```

Sortie

```
-- Moyenne = 14.375
nom     | note
--------+------
Bernard | 15.5
Petit   | 16.0
Moreau  | 17.5
Roux    | 18.0
```

5.7.2 Sous-requête avec IN

Sous-requête avec IN

```sql
-- Etudiants inscrits a un cours du professeur Dupont
SELECT nom, prenom
FROM Etudiants
WHERE id_etudiant IN (
    SELECT id_etudiant
    FROM Inscriptions
    WHERE id_cours IN (
        SELECT id_cours FROM Cours WHERE id_prof = 1
    )
);
```

5.7.3 Sous-requête corrélée

Définition 5.6 (Sous-requête corrélée)**.** Une sous-requête est *corrélée* lorsqu'elle référence une colonne de la requête externe. Elle est réévaluée pour chaque ligne de la requête externe.

Sous-requête corrélée avec EXISTS

```sql
-- Etudiants inscrits a au moins un cours
SELECT e.nom, e.prenom
FROM Etudiants e
WHERE EXISTS (
    SELECT 1
    FROM Inscriptions i
    WHERE i.id_etudiant = e.id_etudiant
);

-- Etudiants NON inscrits a aucun cours
SELECT e.nom, e.prenom
FROM Etudiants e
WHERE NOT EXISTS (
    SELECT 1
    FROM Inscriptions i
    WHERE i.id_etudiant = e.id_etudiant
);
```

Performance des sous-requêtes corrélées

Les sous-requêtes corrélées sont réévaluées pour chaque ligne. Sur de grandes tables, elles peuvent être lentes. Les optimiseurs modernes les transforment souvent en jointures, mais il est bon de connaître l'alternative avec `JOIN` :

```sql
-- Equivalent avec LEFT JOIN + IS NULL
SELECT e.nom FROM Etudiants e
LEFT JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
WHERE i.id_etudiant IS NULL;
```

5.7.4 Sous-requête dans FROM (table dérivée)

Sous-requête dans FROM

```sql
-- Moyenne par etudiant, puis filtrage
SELECT sub.nom, sub.moyenne
FROM (
    SELECT e.nom, AVG(i.note) AS moyenne
    FROM Etudiants e
    JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
    GROUP BY e.id_etudiant, e.nom
) AS sub
WHERE sub.moyenne > 14;
```

Sortie

```
nom      | moyenne
---------+--------
Bernard  | 14.25
Petit    | 14.0
Moreau   | 15.75
Roux     | 18.0
```

5.7.5 Sous-requête dans SELECT

Sous-requête dans SELECT

```sql
SELECT e.nom,
       (SELECT COUNT(*) FROM Inscriptions i
        WHERE i.id_etudiant = e.id_etudiant) AS nb_cours
FROM Etudiants e;
```

Sortie

```
nom      | nb_cours
---------+---------
Bernard  | 2
Petit    | 2
Moreau   | 2
Leroy    | 1
Roux     | 1
```

5.8 Opérateurs ALL, ANY/SOME

Comparaison avec ALL et ANY

```sql
-- Note superieure a TOUTES les notes du cours 101
SELECT e.nom, i.note
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
WHERE i.note > ALL (
    SELECT note FROM Inscriptions WHERE id_cours = 101
);

-- Note superieure a AU MOINS UNE note du cours 101
SELECT e.nom, i.note
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
WHERE i.note > ANY (
    SELECT note FROM Inscriptions WHERE id_cours = 101
);
```

5.9 Expressions de table communes (CTE)

Définition 5.7 (CTE (Common Table Expression))**.** Une *CTE* est une requête nommée temporaire, définie avec le mot-clé `WITH`, qui peut être référencée dans la requête principale. Elle améliore la lisibilité et permet la récursion.

CTE simple

```sql
WITH Moyennes AS (
    SELECT id_etudiant, AVG(note) AS moy
    FROM Inscriptions
    GROUP BY id_etudiant
)
SELECT e.nom, m.moy
FROM Etudiants e
JOIN Moyennes m ON e.id_etudiant = m.id_etudiant
WHERE m.moy > 14
ORDER BY m.moy DESC;
```

CTE récursive — hiérarchie

```sql
-- Table de categories avec parent
CREATE TABLE Categories (
    id INTEGER PRIMARY KEY,
    nom TEXT,
    parent_id INTEGER REFERENCES Categories(id)
);

-- Parcours recursif de la hierarchie
WITH RECURSIVE Arbre AS (
    -- Cas de base : racines
    SELECT id, nom, parent_id, 0 AS niveau
    FROM Categories WHERE parent_id IS NULL

    UNION ALL

    -- Cas recursif
    SELECT c.id, c.nom, c.parent_id, a.niveau + 1
    FROM Categories c
    JOIN Arbre a ON c.parent_id = a.id
)
SELECT * FROM Arbre ORDER BY niveau, nom;
```

5.10 Intégration Python

Jointures et sous-requêtes avec Python

```python
import sqlite3

conn = sqlite3.connect('universite.db')
cur = conn.cursor()

# Jointure : etudiants et leurs cours
cur.execute('''
    SELECT e.nom, c.intitule, i.note
    FROM Etudiants e
    JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
    JOIN Cours c ON i.id_cours = c.id_cours
    WHERE i.note > ?
    ORDER BY i.note DESC
''', (14,))

print("Etudiants avec note > 14 :")
for nom, cours, note in cur.fetchall():
    print(f"  {nom:10} | {cours:20} | {note:.1f}")

# Sous-requete : etudiants au-dessus de la moyenne
cur.execute('''
    SELECT nom, prenom FROM Etudiants
    WHERE id_etudiant IN (
        SELECT id_etudiant FROM Inscriptions
        WHERE note > (SELECT AVG(note) FROM Inscriptions)
    )
''')
print("\nAu-dessus de la moyenne :", cur.fetchall())

conn.close()
```

Exercices

Exercice 5.1. Écrivez une requête qui affiche le nom de chaque professeur et le nombre de cours qu'il enseigne. Incluez les professeurs qui n'enseignent aucun cours.

Exercice 5.2. Trouvez les étudiants qui sont inscrits à tous les cours du professeur dont l'`id_prof` est 2. Utilisez une sous-requête avec `NOT EXISTS`.

Exercice 5.3. Écrivez la même requête que l'exercice précédent en utilisant `GROUP BY` et `HAVING` avec `COUNT`. Comparez les deux approches.

Exercice 5.4. Trouvez les paires d'étudiants qui ont au moins deux cours en commun. Affichez les noms des deux étudiants et le nombre de cours partagés.

Exercice 5.5. En utilisant une CTE récursive, générez la table des nombres de 1 à 20 et affichez leur factorielle.

CHAPITRE 5. SQL — JOINTURES ET SOUS-REQUÊTES
CHAPITRE 5. SQL — JOINTURES ET SOUS-REQUÊTES

Chapitre 6

SQL — Agrégation et Fonctions de Fenêtre

Jusqu'ici, nos requêtes SQL retournaient des lignes individuelles. Mais souvent, on veut des résumés : le nombre total d'étudiants, la moyenne des notes par département, le chiffre d'affaires par trimestre. C'est le rôle des *fonctions d'agrégation* (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`) combinées avec `GROUP BY`. Les *fonctions de fenêtre* (`OVER`), introduites dans SQL :2003, vont plus loin : elles calculent des agrégats tout en conservant les lignes individuelles — rang, cumul, moyenne mobile — sans nécessiter de sous-requêtes complexes.

6.1 Fonctions d'agrégation

Définition 6.1 (Fonction d'agrégation)**.** Une *fonction d'agrégation* opère sur un ensemble de lignes et retourne une valeur unique résumant cet ensemble.

Fonctions d'agrégation standard

Fonction	Description	Ignore NULL ?
`COUNT(*)`	Nombre de lignes	Non
`COUNT(col)`	Nombre de valeurs non NULL	Oui
`COUNT(DISTINCT col)`	Nombre de valeurs distinctes	Oui
`SUM(col)`	Somme	Oui
`AVG(col)`	Moyenne	Oui
`MIN(col)`	Minimum	Oui
`MAX(col)`	Maximum	Oui

Agrégations simples

```
SELECT COUNT(*) AS nb_inscriptions,
       COUNT(DISTINCT id_etudiant) AS nb_etudiants,
       AVG(note) AS moyenne,
       MIN(note) AS note_min,
       MAX(note) AS note_max,
       SUM(note) AS somme_notes
FROM Inscriptions;
```

Sortie

```
nb_inscriptions | nb_etudiants | moyenne | note_min | note_max | somme_notes
----------------+--------------+---------+----------+----------+------------
8               | 5            | 14.375  | 9.0      | 18.0     | 115.0
```

AVG et les valeurs NULL

`AVG` ignore les valeurs `NULL`. La moyenne de $\{10, \texttt{NULL}, 20\}$ est 15, pas 10. Si vous souhaitez traiter les NULL comme 0, utilisez `AVG(COALESCE(note, 0))`.

6.2 GROUP BY — Regroupement

Définition 6.2 (GROUP BY)**.** La clause `GROUP BY` partitionne les lignes en groupes selon une ou plusieurs colonnes. Les fonctions d'agrégation sont alors appliquées à chaque groupe séparément.

Ordre d'exécution avec GROUP BY

1. `FROM` / `JOIN`
2. `WHERE` (filtre avant regroupement)
3. `GROUP BY`
4. `HAVING` (filtre après regroupement)
5. `SELECT`
6. `DISTINCT`
7. `ORDER BY`
8. `LIMIT`

Moyenne par étudiant

```sql
SELECT e.nom, e.prenom,
       COUNT(*) AS nb_cours,
       ROUND(AVG(i.note), 2) AS moyenne,
       MIN(i.note) AS note_min,
       MAX(i.note) AS note_max
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
GROUP BY e.id_etudiant, e.nom, e.prenom
ORDER BY moyenne DESC;
```

Sortie

```
nom      | prenom | nb_cours | moyenne | note_min | note_max
---------+--------+----------+---------+----------+---------
```

```
Roux    | Emma   | 1        | 18.0    | 18.0     | 18.0
Moreau  | Claire | 2        | 15.75   | 14.0     | 17.5
Bernard | Alice  | 2        | 14.25   | 13.0     | 15.5
Petit   | Bob    | 2        | 14.0    | 12.0     | 16.0
Leroy   | David  | 1        | 9.0     | 9.0      | 9.0
```

Colonne hors GROUP BY sans agrégation

Toute colonne dans le `SELECT` doit soit apparaître dans le `GROUP BY`, soit être dans une fonction d'agrégation :

```sql
-- ERREUR : prenom n'est ni dans GROUP BY ni agrege
SELECT nom, prenom, AVG(note) FROM Inscriptions
JOIN Etudiants ON ... GROUP BY nom;

-- CORRECT :
SELECT nom, prenom, AVG(note) FROM Inscriptions
JOIN Etudiants ON ... GROUP BY nom, prenom;
```

MySQL en mode non strict accepte cette erreur silencieusement (résultat indéterminé). PostgreSQL et SQLite la rejettent correctement.

6.3 HAVING — Filtrage après regroupement

Définition 6.3 (HAVING)**.** La clause `HAVING` filtre les groupes après agrégation. Elle est à `GROUP BY` ce que `WHERE` est à `FROM` : `WHERE` filtre les lignes, `HAVING` filtre les groupes.

Filtrage avec HAVING

```sql
-- Etudiants ayant une moyenne superieure a 14
SELECT e.nom, AVG(i.note) AS moyenne
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
GROUP BY e.id_etudiant, e.nom
HAVING AVG(i.note) > 14
ORDER BY moyenne DESC;

-- Cours avec au moins 2 inscrits
SELECT c.intitule, COUNT(*) AS nb_inscrits
FROM Cours c
JOIN Inscriptions i ON c.id_cours = i.id_cours
GROUP BY c.id_cours, c.intitule
HAVING COUNT(*) >= 2;
```

Sortie

```
-- Moyenne > 14 :
nom     | moyenne
--------+--------
Roux    | 18.0
Moreau  | 15.75
Bernard | 14.25

-- Cours avec >= 2 inscrits :
intitule         | nb_inscrits
-----------------+------------
Bases de donnees | 3
Algorithmes      | 2
Statistiques     | 2
```

6.4 Groupements avancés

6.4.1 GROUPING SETS, ROLLUP, CUBE

ROLLUP — sous-totaux hiérarchiques (PostgreSQL)

```sql
SELECT e.filiere,
       c.intitule,
       AVG(i.note) AS moyenne,
       COUNT(*) AS nb
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
JOIN Cours c ON i.id_cours = c.id_cours
GROUP BY ROLLUP (e.filiere, c.intitule)
ORDER BY e.filiere, c.intitule;
```

`ROLLUP` génère les sous-totaux hiérarchiques : par (filière, cours), par filière, et le total général.

`CUBE` génère toutes les combinaisons possibles de sous-totaux.

6.5 Fonctions de fenêtre (Window Functions)

Définition 6.4 (Fonction de fenêtre)**.** Une *fonction de fenêtre* effectue un calcul sur un ensemble de lignes liées à la ligne courante, *sans réduire le nombre de lignes* du résultat. Elle utilise la clause `OVER()`.

Théorème 6.5 (Différence GROUP BY vs. Window Function)**.**

- *`GROUP BY` + agrégation : réduit les lignes (une ligne par groupe).*
- *Fonction de fenêtre : conserve toutes les lignes tout en ajoutant une valeur calculée.*

6.5.1 Syntaxe OVER

Syntaxe des fonctions de fenêtre

```
fonction(args) OVER (
    [PARTITION BY colonnes]
    [ORDER BY colonnes [ASC|DESC]]
    [ROWS BETWEEN debut AND fin]
)
```

6.5.2 Agrégation fenêtrée

Moyenne fenêtrée par filière

```
SELECT e.nom, e.filiere, i.note,
       AVG(i.note) OVER (PARTITION BY e.filiere) AS moy_filiere,
       i.note - AVG(i.note) OVER (PARTITION BY e.filiere) AS ecart
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
ORDER BY e.filiere, e.nom;
```

Sortie

```
nom     | filiere | note | moy_filiere | ecart
--------+---------+------+-------------+------
Bernard | Info    | 15.5 | 13.1        | 2.4
Bernard | Info    | 13.0 | 13.1        | -0.1
Leroy   | Info    | 9.0  | 13.1        | -4.1
Petit   | Info    | 12.0 | 13.1        | -1.1
Petit   | Info    | 16.0 | 13.1        | 2.9
Moreau  | Maths   | 17.5 | 16.5        | 1.0
Moreau  | Maths   | 14.0 | 16.5        | -2.5
Roux    | Maths   | 18.0 | 16.5        | 1.5
```

6.5.3 ROW_NUMBER, RANK, DENSE_RANK

Définition 6.6 (Fonctions de classement).

- `ROW_NUMBER()` — numéro séquentiel unique dans la partition.
- `RANK()` — rang avec trous (ex-aequo reçoivent le même rang, le rang suivant est incrémenté du nombre d'ex-aequo).
- `DENSE_RANK()` — rang sans trous.

Classement des étudiants par note

```
SELECT e.nom, i.note,
       ROW_NUMBER() OVER (ORDER BY i.note DESC) AS row_num,
```

```
       RANK()       OVER (ORDER BY i.note DESC) AS rang,
       DENSE_RANK() OVER (ORDER BY i.note DESC) AS rang_dense
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant;
```

Sortie

```
nom     | note | row_num | rang | rang_dense
--------+------+---------+------+-----------
Roux    | 18.0 | 1       | 1    | 1
Moreau  | 17.5 | 2       | 2    | 2
Petit   | 16.0 | 3       | 3    | 3
Bernard | 15.5 | 4       | 4    | 4
Moreau  | 14.0 | 5       | 5    | 5
Bernard | 13.0 | 6       | 6    | 6
Petit   | 12.0 | 7       | 7    | 7
Leroy   | 9.0  | 8       | 8    | 8
```

Meilleur étudiant par filière (Top-N par groupe)

```
WITH Classement AS (
    SELECT e.nom, e.filiere, i.note,
           ROW_NUMBER() OVER (
               PARTITION BY e.filiere
               ORDER BY i.note DESC
           ) AS rn
    FROM Etudiants e
    JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
)
SELECT nom, filiere, note
FROM Classement
WHERE rn = 1;
```

6.5.4 LAG, LEAD

Définition 6.7 (LAG et LEAD).

- `LAG(col, n, defaut)` — valeur de `col` n lignes *avant* la ligne courante.
- `LEAD(col, n, defaut)` — valeur de `col` n lignes *après* la ligne courante.

Comparaison avec la ligne précédente

```
SELECT e.nom, i.note,
       LAG(i.note, 1) OVER (ORDER BY i.note DESC) AS note_precedente,
       i.note - LAG(i.note, 1) OVER (ORDER BY i.note DESC) AS diff
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
ORDER BY i.note DESC;
```

6.5.5 Fenêtres glissantes (Frame)

Moyenne mobile sur 3 lignes

```
SELECT e.nom, i.note,
       AVG(i.note) OVER (
           ORDER BY i.note
           ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
       ) AS moyenne_mobile
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
ORDER BY i.note;
```

Spécifications de fenêtre (Frame)

Spécification	Description
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	Du début à la ligne courante
`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`	Fenêtre glissante de 3 lignes
`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`	De la ligne courante à la fin
`RANGE BETWEEN ...`	Basé sur les valeurs (pas les positions)

6.5.6 NTILE et calculs de pourcentage

Quartiles et pourcentage cumulatif

```
SELECT e.nom, i.note,
       NTILE(4) OVER (ORDER BY i.note DESC) AS quartile,
       ROUND(100.0 * SUM(i.note) OVER (ORDER BY i.note DESC
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
           / SUM(i.note) OVER (), 1) AS pct_cumul
FROM Etudiants e
JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
ORDER BY i.note DESC;
```

6.6 Intégration Python

Agrégation et window functions avec Python

```
import sqlite3

conn = sqlite3.connect('universite.db')
cur = conn.cursor()

# Agregation avec GROUP BY
cur.execute('''
```

```
    SELECT e.nom, COUNT(*) as nb, ROUND(AVG(i.note), 2) as moy
    FROM Etudiants e
    JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
    GROUP BY e.id_etudiant, e.nom
    HAVING AVG(i.note) > 14
    ORDER BY moy DESC
''')
print("Moyennes > 14 :")
for row in cur.fetchall():
    print(f"  {row[0]:10} | {row[1]} cours | moyenne {row[2]}")

# Window function (SQLite >= 3.25)
cur.execute('''
    SELECT e.nom, i.note,
           RANK() OVER (ORDER BY i.note DESC) as rang
    FROM Etudiants e
    JOIN Inscriptions i ON e.id_etudiant = i.id_etudiant
''')
print("\nClassement :")
for nom, note, rang in cur.fetchall():
    print(f"  #{rang} {nom:10} : {note}")

conn.close()
```

Exercices

Exercice 6.1. Pour chaque cours, affichez le nombre d'inscrits, la moyenne, la note minimale et maximale. N'affichez que les cours ayant au moins 2 inscrits.

Exercice 6.2. Classez les étudiants par moyenne générale et affichez leur rang. Utilisez `DENSE_RANK` pour gérer les ex-aequo.

Exercice 6.3. Pour chaque inscription, affichez la note de l'étudiant, la moyenne de sa filière, et l'écart par rapport à cette moyenne. Utilisez une fonction de fenêtre.

Exercice 6.4. Affichez le top-2 des meilleures notes par cours. Utilisez `ROW_NUMBER` avec `PARTITION BY`.

Exercice 6.5. Calculez la somme cumulative des notes par étudiant, ordonnée par identifiant de cours, en utilisant une fenêtre glissante.

Chapitre 7

Normalisation des Bases de Données

Pourquoi une base de données bien conçue ne contient-elle jamais de redondance ? La réponse tient en un mot : les *anomalies*. Quand la même information est stockée à plusieurs endroits, une mise à jour partielle crée des incohérences ; une suppression peut détruire des données que l'on voulait conserver ; une insertion peut être impossible sans inventer des valeurs fictives. Edgar Codd, dès 1972, a formalisé ces problèmes en introduisant les *dépendances fonctionnelles* et les *formes normales*. L'idée est systématique : décomposer les tables pour éliminer la redondance tout en préservant l'information (décomposition sans perte) et les contraintes (préservation des dépendances). La première forme normale (1NF), la deuxième (2NF), la troisième (3NF), et la forme normale de Boyce–Codd (BCNF) forment une hiérarchie de plus en plus stricte. Comprendre cette hiérarchie, c'est comprendre l'art de concevoir des schémas relationnels robustes.

7.1 Problèmes de conception et anomalies

Un schéma mal conçu entraîne des *anomalies* :

Définition 7.1 (Anomalies de données)**.**

- **Anomalie d'insertion** : impossibilité d'insérer des données sans en insérer d'autres non souhaitées.
- **Anomalie de mise à jour** : une modification doit être répétée sur plusieurs lignes pour maintenir la cohérence.
- **Anomalie de suppression** : la suppression d'une information entraîne la perte involontaire d'autres informations.

Exemple 7.2. Considérons la table non normalisée suivante :

id_et	nom	cours	prof	dept_prof	note
1	Bernard	BDD	Dupont	Info	15.5
1	Bernard	Algo	Martin	Maths	13.0
2	Petit	BDD	Dupont	Info	12.0

- **Mise à jour** : si Dupont change de département, il faut modifier *toutes* les lignes où il apparaît.
- **Insertion** : on ne peut pas enregistrer un nouveau cours sans inscrire un étudiant.
- **Suppression** : supprimer l'inscription de Petit à BDD fait disparaître l'information sur le cours BDD si c'est la dernière ligne.

7.2 Dépendances fonctionnelles

Définition 7.3 (Dépendance fonctionnelle)**.** Soit R une relation et $X, Y \subseteq \text{attr}(R)$. On dit que Y dépend fonctionnellement de X, noté $X \to Y$, si pour tout couple de tuples $t_1, t_2 \in R$:

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$$

Autrement dit, la valeur de X détermine de manière unique la valeur de Y.

Exemple 7.4. Dans la table Université :

- id_etudiant $\to$ nom, prenom, filiere
- id_cours $\to$ intitule, credits, id_prof
- id_prof $\to$ nom_prof, departement
- (id_etudiant, id_cours) $\to$ note

Définition 7.5 (Dépendance fonctionnelle triviale)**.** $X \to Y$ est *triviale* si $Y \subseteq X$.

Définition 7.6 (Dépendance fonctionnelle partielle et transitive)**.**

- **Partielle** : $X \to Y$ est partielle s'il existe $X' \subsetneq X$ tel que $X' \to Y$.
- **Transitive** : si $X \to Z$ et $Z \to Y$ (avec Z non clé), alors $X \to Y$ est transitive.

7.3 Axiomes d'Armstrong

Théorème 7.7 (Axiomes d'Armstrong)**.** *Les axiomes d'Armstrong sont* corrects *(sound) et* complets *:*

1. ***Réflexivité** : si $Y \subseteq X$, alors $X \to Y$.*
2. ***Augmentation** : si $X \to Y$, alors $XZ \to YZ$.*
3. ***Transitivité** : si $X \to Y$ et $Y \to Z$, alors $X \to Z$.*

Règles dérivées :

- ***Union** : si $X \to Y$ et $X \to Z$, alors $X \to YZ$.*
- ***Décomposition** : si $X \to YZ$, alors $X \to Y$ et $X \to Z$.*
- ***Pseudo-transitivité** : si $X \to Y$ et $WY \to Z$, alors $WX \to Z$.*

7.4 Fermeture d'un ensemble d'attributs

Définition 7.8 (Fermeture X^+)**.** La *fermeture* de X par rapport à un ensemble de DF F, notée X_F^+, est l'ensemble de tous les attributs A tels que $X \to A$ peut être déduit de F par les axiomes d'Armstrong.

Proposition 7.9 (Algorithme de fermeture)**.** Pour calculer X^+ :

1. Initialiser $X^+ = X$.

2. Répéter : pour chaque DF $Y \to Z$ dans F, si $Y \subseteq X^+$, ajouter Z à X^+.
3. S'arrêter quand X^+ ne change plus.

Exemple 7.10. Soit $R(A, B, C, D, E)$ avec $F = \{A \to B,\ BC \to D,\ D \to E\}$.
Calculons $\{A, C\}^+$:

1. $X^+ = \{A, C\}$
2. $A \to B$ et $A \in X^+ : X^+ = \{A, B, C\}$
3. $BC \to D$ et $\{B, C\} \subseteq X^+ : X^+ = \{A, B, C, D\}$
4. $D \to E$ et $D \in X^+ : X^+ = \{A, B, C, D, E\}$

Comme X^+ contient tous les attributs, $\{A, C\}$ est une clé candidate.

7.5 Première forme normale (1NF)

Définition 7.11 (1NF)**.** Une relation est en *première forme normale* (1NF) si tous ses attributs sont *atomiques* : chaque cellule contient une valeur unique et indivisible.

Exemple 7.12. Table NON 1NF :

id	nom	telephones
1	Bernard	0612345678, 0198765432

Table 1NF (après décomposition) :

id	nom
1	Bernard

id	telephone
1	0612345678
1	0198765432

7.6 Deuxième forme normale (2NF)

Définition 7.13 (2NF)**.** Une relation est en *deuxième forme normale* si elle est en 1NF et tout attribut non-clé dépend *totalement* de la clé primaire (pas de dépendance fonctionnelle partielle).

Remarque 7.14. La 2NF n'est pertinente que pour les relations ayant une clé composée. Si la clé est un attribut unique, la relation en 1NF est automatiquement en 2NF.

Exemple 7.15. Soit $R(\underline{\text{id\_et}}, \underline{\text{id\_cours}}, \text{nom\_et}, \text{note})$ avec la clé (id_et, id_cours).
id_et $\to$ nom_et est une DF partielle (ne dépend que d'une partie de la clé). La relation n'est pas en 2NF.
Décomposition :

- $R_1(\underline{\text{id\_et}}, \text{nom\_et})$
- $R_2(\underline{\text{id\_et}}, \underline{\text{id\_cours}}, \text{note})$

7.7 Troisième forme normale (3NF)

Définition 7.16 (3NF)**.** Une relation est en *troisième forme normale* si elle est en 2NF et aucun attribut non-clé ne dépend transitivement de la clé primaire.

Formellement, pour toute DF non triviale $X \to A$:

- soit X contient une clé candidate (super-clé),
- soit A est un attribut premier (appartient à une clé candidate).

Exemple 7.17. Soit $R(\underline{\text{id\_et}}, \text{filiere}, \text{responsable\_filiere})$.

DF : id_et $\to$ filiere et filiere $\to$ responsable_filiere.

La DF id_et $\to$ responsable_filiere est transitive. Pas en 3NF.

Décomposition :

- $R_1(\underline{\text{id\_et}}, \text{filiere})$
- $R_2(\underline{\text{filiere}}, \text{responsable\_filiere})$

7.8 Forme normale de Boyce-Codd (BCNF)

Définition 7.18 (BCNF)**.** Une relation est en *BCNF* si, pour toute dépendance fonctionnelle non triviale $X \to Y$, X est une super-clé.

Théorème 7.19 (BCNF vs. 3NF)**.**

- *BCNF* $\Rightarrow$ *3NF* $\Rightarrow$ *2NF* $\Rightarrow$ *1NF.*
- *BCNF est plus stricte que 3NF : elle interdit même les dépendances où le déterminant n'est pas une super-clé, même si le déterminé est un attribut premier.*
- *La décomposition en BCNF ne préserve pas toujours les DF.*
- *La décomposition en 3NF préserve toujours les DF.*

Exemple 7.20. Soit $R(\underline{A}, \underline{B}, C)$ avec les DF : $AB \to C$ et $C \to A$.

$C \to A$: C n'est pas super-clé (la clé est $\{A, B\}$ ou $\{B, C\}$). Donc pas en BCNF.

Mais A est premier (fait partie de la clé $\{A, B\}$). Donc en 3NF.

7.9 Décomposition sans perte

Théorème 7.21 (Décomposition sans perte (Lossless Join))**.** *Une décomposition de R en R_1 et R_2 est* sans perte *si et seulement si l'un des DF suivants est vérifié :*

$$R_1 \cap R_2 \to R_1 - R_2 \quad ou \quad R_1 \cap R_2 \to R_2 - R_1$$

c'est-à-dire si l'intersection des schémas détermine au moins l'un des compléments.

7.10 Algorithme de décomposition en BCNF

Proposition 7.22 (Algorithme BCNF)**.**

1. Vérifier si la relation est en BCNF.
2. Sinon, trouver une DF $X \to Y$ violant BCNF (où X n'est pas super-clé).
3. Décomposer en :
   - $R_1 = X \cup Y$ (avec clé X)
   - $R_2 = R - Y \cup X$ (tous les attributs sauf Y, mais incluant X)
4. Répéter sur R_1 et R_2 si nécessaire.

7.11 Résumé des formes normales

Synthèse des formes normales

FN	Condition
1NF	Tous les attributs sont atomiques
2NF	1NF + pas de DF partielle d'un attribut non-clé par rapport à la clé
3NF	2NF + pas de DF transitive d'un attribut non-clé par rapport à la clé
BCNF	Pour toute DF $X \to Y$ non triviale, X est une super-clé

7.12 Synthèse 3NF (algorithme de Bernstein)

Proposition 7.23 (Algorithme de synthèse 3NF)**.**

1. Calculer une couverture minimale $F_{\min}$ des DF.
2. Pour chaque DF $X \to Y$ dans $F_{\min}$, créer une relation $R_i(X, Y)$.
3. Si aucune relation ne contient une clé candidate de R, ajouter une relation avec une clé candidate.
4. Éliminer les relations redondantes (incluses dans une autre).

Cet algorithme garantit :

- Décomposition sans perte.
- Préservation des dépendances fonctionnelles.
- Résultat en 3NF.

7.13 Exemple complet de normalisation

Table non normalisée

```sql
-- Table avec redondances
CREATE TABLE Inscriptions_Denorm (
    id_etudiant INTEGER,
    nom_etudiant TEXT,
    filiere TEXT,
    responsable_filiere TEXT,
    id_cours INTEGER,
    intitule_cours TEXT,
    id_prof INTEGER,
    nom_prof TEXT,
    note REAL
);
```

DF identifiées :

- id_et $\rightarrow$ nom_et, filiere
- filiere $\rightarrow$ responsable_filiere
- id_cours $\rightarrow$ intitule, id_prof
- id_prof $\rightarrow$ nom_prof
- (id_et, id_cours) $\rightarrow$ note

Résultat après normalisation en 3NF/BCNF

```sql
CREATE TABLE Etudiants (
    id_etudiant INTEGER PRIMARY KEY,
    nom TEXT NOT NULL,
    filiere TEXT REFERENCES Filieres(filiere)
);

CREATE TABLE Filieres (
    filiere TEXT PRIMARY KEY,
    responsable TEXT NOT NULL
);

CREATE TABLE Professeurs (
    id_prof INTEGER PRIMARY KEY,
    nom TEXT NOT NULL
);

CREATE TABLE Cours (
    id_cours INTEGER PRIMARY KEY,
    intitule TEXT NOT NULL,
    id_prof INTEGER REFERENCES Professeurs(id_prof)
);
```

```sql
CREATE TABLE Inscriptions (
    id_etudiant INTEGER REFERENCES Etudiants(id_etudiant),
    id_cours INTEGER REFERENCES Cours(id_cours),
    note REAL,
    PRIMARY KEY (id_etudiant, id_cours)
);
```

7.14 Dénormalisation contrôlée

Remarque 7.24. La normalisation parfaite n'est pas toujours le meilleur choix en pratique. La *dénormalisation* consiste à réintroduire volontairement de la redondance pour améliorer les performances de lecture (éviter des jointures coûteuses).

Quand dénormaliser ?

- Quand les lectures sont beaucoup plus fréquentes que les écritures.
- Pour des tables de reporting ou d'entrepôt de données.
- Toujours documenter et justifier la dénormalisation.
- Utiliser des triggers ou des vues matérialisées pour maintenir la cohérence.

Exercices

Exercice 7.1. Soit $R(A, B, C, D, E)$ avec $F = \{A \to BC,\ C \to D,\ BD \to E\}$.

1. Calculez $\{A\}^+$.
2. Identifiez les clés candidates.
3. La relation est-elle en 2NF ? 3NF ? BCNF ? Justifiez.
4. Décomposez en BCNF.

Exercice 7.2. Normalisez la relation suivante en 3NF en utilisant l'algorithme de synthèse : R(NumCommande, DateCmd, NumClient, NomClient, NumProduit, NomProduit, Quantite, Prix
DF : NumCmd $\to$ DateCmd, NumClient ; NumClient $\to$ NomClient ; NumProduit $\to$ NomProduit, PrixUnit ; (NumCmd, NumProduit) $\to$ Quantite.

Exercice 7.3. Donnez un exemple de relation qui est en 3NF mais pas en BCNF. Décomposez-la en BCNF et vérifiez si les dépendances fonctionnelles sont préservées.

Exercice 7.4. Vérifiez que la décomposition de $R(A, B, C)$ avec $F = \{A \to B\}$ en $R_1(A, B)$ et $R_2(A, C)$ est sans perte. Utilisez le théorème de Heath.

Chapitre 8

Transactions et Contrôle de Concurrence

Imaginez un virement bancaire : on débite un compte et on crédite un autre. Que se passe-t-il si le système tombe en panne entre les deux opérations ? L'argent a disparu. C'est pour éviter ce genre de catastrophe que Jim Gray, dans les années 1970 chez IBM, a formalisé la notion de *transaction* et les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité). Une transaction est un "tout ou rien" logique : soit toutes ses opérations réussissent, soit aucune n'a d'effet. Quand plusieurs transactions s'exécutent simultanément, le contrôle de concurrence (verrouillage, estampillage, MVCC) garantit que le résultat est équivalent à une exécution séquentielle. Gray recevra le prix Turing en 1998 pour ces travaux fondamentaux.

Idée centrale

Une transaction est une unité logique de travail qui doit être exécutée de manière atomique, cohérente, isolée et durable (ACID). Le contrôle de concurrence garantit que l'exécution simultanée de plusieurs transactions produit un résultat correct, comme si elles avaient été exécutées séquentiellement.

8.1 Notion de transaction

Définition 8.1 (Transaction)**.** Une **transaction** est une séquence d'opérations de lecture et d'écriture sur la base de données, délimitée par un `BEGIN` et terminée par un `COMMIT` (validation) ou un `ROLLBACK` (annulation).

```python
# Exemple en Python avec psycopg2
import psycopg2

conn = psycopg2.connect("dbname=banque")
cur = conn.cursor()
try:
    cur.execute("UPDATE comptes SET solde = solde - 100 WHERE id = 1")
    cur.execute("UPDATE comptes SET solde = solde + 100 WHERE id = 2")
    conn.commit()   # Les deux operations sont validees ensemble
except Exception:
```

```
    conn.rollback()  # Aucune modification n'est appliquee
finally:
    cur.close()
    conn.close()
```

8.2 Propriétés ACID

Définition 8.2 (Propriétés ACID)**.** Les quatre propriétés fondamentales d'une transaction sont :

1. **Atomicité** — Toutes les opérations sont exécutées ou aucune.
2. **Cohérence** — La transaction amène la base d'un état cohérent à un autre état cohérent.
3. **Isolation** — Les transactions concurrentes ne voient pas les modifications intermédiaires des autres.
4. **Durabilité** — Une fois validée (`COMMIT`), les modifications survivent à toute panne.

Remarque 8.3. L'atomicité et la durabilité sont assurées par le journal de transactions (*Write-Ahead Log*, WAL). La cohérence est assurée par les contraintes d'intégrité. L'isolation est assurée par le contrôle de concurrence.

8.3 Problèmes liés à la concurrence

Sans contrôle, l'exécution concurrente peut produire des anomalies :

Définition 8.4 (Anomalies de concurrence)**.**

- **Lecture sale** (*dirty read*) — T_2 lit une donnée modifiée par T_1 qui n'a pas encore validé.
- **Lecture non répétable** (*non-repeatable read*) — T_1 lit une donnée deux fois et obtient des valeurs différentes car T_2 l'a modifiée entre-temps.
- **Lecture fantôme** (*phantom read*) — T_1 exécute une requête deux fois et obtient des lignes différentes car T_2 a inséré ou supprimé des lignes.
- **Écriture perdue** (*lost update*) — Deux transactions lisent la même valeur, la modifient et l'écrivent : une des modifications est perdue.

8.4 Niveaux d'isolation

Définition 8.5 (Niveaux d'isolation SQL)**.** Le standard SQL définit quatre niveaux d'isolation :

Niveau	Dirty read	Non-repeatable	Phantom
`READ UNCOMMITTED`	Possible	Possible	Possible
`READ COMMITTED`	Non	Possible	Possible
`REPEATABLE READ`	Non	Non	Possible
`SERIALIZABLE`	Non	Non	Non

Attention

Le niveau par défaut varie selon le SGBD : `READ COMMITTED` pour PostgreSQL et Oracle, `REPEATABLE READ` pour MySQL/InnoDB. Choisir un niveau trop faible expose aux anomalies ; un niveau trop fort réduit les performances.

```
-- Fixer le niveau d'isolation en SQL
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
    SELECT solde FROM comptes WHERE id = 1;
    UPDATE comptes SET solde = solde - 100 WHERE id = 1;
COMMIT;
```

8.5 Théorie de la sérialisabilité

Définition 8.6 (Ordonnancement (schedule))**.** Un **ordonnancement** S est une séquence d'opérations (lectures, écritures, commits, rollbacks) provenant de plusieurs transactions, qui préserve l'ordre interne de chaque transaction.

Définition 8.7 (Sérialisabilité par conflit)**.** Deux opérations sont en **conflit** si elles portent sur le même objet et au moins l'une est une écriture. Un ordonnancement S est **sérialisable par conflit** s'il est équivalent par conflit à un ordonnancement sériel.

Théorème 8.8 (Test de sérialisabilité par graphe de précédence)**.** *Construire le* ***graphe de précédence*** $G = (V, E)$ *où* V *est l'ensemble des transactions et* $(T_i, T_j) \in E$ *si une opération de* T_i *est en conflit avec une opération de* T_j *et la précède. L'ordonnancement est sérialisable par conflit si et seulement si* G *est acyclique.*

Exemple 8.9. Soit S : $r_1(A)$, $w_1(A)$, $r_2(A)$, $w_2(A)$, $r_1(B)$, $w_1(B)$, $r_2(B)$, $w_2(B)$. Les conflits sont : $w_1(A) \to r_2(A)$, $w_1(A) \to w_2(A)$, $w_1(B) \to r_2(B)$, $w_1(B) \to w_2(B)$. Le graphe a une seule arête $T_1 \to T_2$, donc S est sérialisable (équivalent à T_1, T_2).

8.6 Verrouillage à deux phases (2PL)

Définition 8.10 (Protocole 2PL)**.** Le protocole de **verrouillage à deux phases** (2PL) impose que chaque transaction passe par deux phases :

1. **Phase de croissance** : la transaction acquiert des verrous (partagés pour lecture, exclusifs pour écriture) sans en libérer.
2. **Phase de décroissance** : la transaction libère des verrous sans en acquérir de nouveaux.

Théorème 8.11 (Correction de 2PL)**.** *Tout ordonnancement produit par le protocole 2PL est sérialisable par conflit.*

Définition 8.12 (2PL strict et rigoureux)**.**

- **2PL strict** : les verrous exclusifs ne sont libérés qu'au `COMMIT`/`ROLLBACK`. Évite les annulations en cascade.
- **2PL rigoureux** : *tous* les verrous sont libérés au `COMMIT`/`ROLLBACK`. C'est le mode le plus courant.

8.7 Détection et prévention des interblocages

Définition 8.13 (Interblocage (deadlock))**.** Un **interblocage** se produit lorsque deux ou plusieurs transactions s'attendent mutuellement, chacune détenant un verrou demandé par l'autre.

Proposition 8.14 (Détection par graphe d'attente)**.** Construire le **graphe d'attente** : un arc (T_i, T_j) signifie que T_i attend un verrou détenu par T_j. Il y a interblocage si et seulement si le graphe contient un cycle.

Stratégies de résolution :

- **Détection périodique** : vérifier les cycles et annuler la transaction la moins coûteuse (*victim selection*).
- **Prévention par estampilles** : *Wait-Die* ou *Wound-Wait*.
- **Timeout** : annuler toute transaction qui attend trop longtemps.

8.8 Contrôle de concurrence multi-versions (MVCC)

Définition 8.15 (MVCC)**.** Le **MVCC** (*Multi-Version Concurrency Control*) maintient plusieurs versions de chaque ligne. Une lecture ne bloque jamais une écriture et vice versa :

- Chaque transaction voit un *snapshot* cohérent de la base au moment de son début.
- Les écritures créent de nouvelles versions sans écraser les anciennes.
- Le ramasse-miettes (*vacuum*) supprime les versions obsolètes.

Remarque 8.16. PostgreSQL utilise MVCC pour tous les niveaux d'isolation. MySQL/InnoDB utilise MVCC pour `READ COMMITTED` et `REPEATABLE READ`, et des verrous supplémentaires pour `SERIALIZABLE`.

Exemple 8.17. Sous MVCC, la transaction T_1 peut lire la version v_1 d'une ligne pendant que T_2 écrit la version v_2 : aucun verrouillage n'est nécessaire pour la lecture.

```
-- PostgreSQL : observer le MVCC
BEGIN;
SELECT xmin, xmax, * FROM comptes WHERE id = 1;
-- xmin = ID de la transaction qui a cree cette version
-- xmax = ID de la transaction qui l'a supprimee (0 si active)
COMMIT;
```

8.9 Journal de transactions (WAL)

Définition 8.18 (Write-Ahead Logging)**.** Le **WAL** (*Write-Ahead Log*) est un journal séquentiel où toute modification est enregistrée *avant* d'être appliquée aux pages de données. En cas de panne, le journal permet :

- **REDO** : rejouer les modifications des transactions validées.
- **UNDO** : annuler les modifications des transactions non validées.

8.10 Formules clés

Formules clés

- **ACID** : Atomicité, Cohérence, Isolation, Durabilité
- **Sérialisabilité** : graphe de précédence acyclique $\Leftrightarrow$ sérialisable
- **2PL** : phase de croissance $\rightarrow$ point de verrouillage $\rightarrow$ phase de décroissance
- **Deadlock** : cycle dans le graphe d'attente
- **MVCC** : lectures non bloquantes via les snapshots multi-versions

8.11 Exercices

Exercice 8.1. Soit l'ordonnancement $S : r_1(A)$, $r_2(B)$, $w_1(B)$, $w_2(A)$. Construire le graphe de précédence. S est-il sérialisable ?

Exercice 8.2. Donner un exemple d'interblocage entre deux transactions portant sur deux comptes bancaires. Proposer une solution utilisant un ordre fixe d'acquisition des verrous.

Exercice 8.3. Écrire un script Python qui démontre une lecture fantôme sous `READ COMMITTED` dans PostgreSQL, puis montrer qu'elle disparaît sous `SERIALIZABLE`.

Exercice 8.4. Expliquer pourquoi le 2PL strict évite les annulations en cascade. Donner un contre-exemple avec le 2PL de base.

Exercice 8.5. Dans un système MVCC, une transaction T_1 lit la ligne r (version v_1), puis T_2 modifie r et valide (version v_2), puis T_1 relit r. Quelle version voit T_1 sous `READ COMMITTED` ? Sous `REPEATABLE READ` ? Justifier.

Chapitre 9

Indexation et Structures de Données

Imaginez une bibliothèque de dix millions de livres sans catalogue. Pour trouver un ouvrage, il faudrait parcourir chaque étagère, un livre à la fois. C'est exactement ce que fait un SGBD lors d'un *full table scan* : il examine chaque ligne de la table. Un *index* est le catalogue de la base de données : une structure auxiliaire qui permet de localiser les lignes pertinentes en $O(\log n)$ au lieu de $O(n)$. L'arbre B+ (B+ tree), inventé par Rudolf Bayer et Edward McCreight en 1972, est la structure d'indexation dominante : équilibré, efficace pour les recherches par égalité et par intervalle, et optimisé pour les accès disque. Les index hash, les index bitmap et les index spatiaux (R-tree) complètent l'arsenal. Ce chapitre développe ces structures et les stratégies d'indexation qui font la différence entre une requête en millisecondes et une requête en minutes.

Idée centrale

Un index est une structure de données auxiliaire qui accélère la recherche dans une table en évitant le parcours séquentiel (*full table scan*). Le choix du bon type d'index et des bonnes colonnes à indexer est essentiel pour les performances.

9.1 Motivation

Sans index, toute recherche dans une table de n lignes nécessite un parcours séquentiel en $O(n)$. Un index bien choisi réduit la complexité à $O(\log n)$ voire $O(1)$.

Remarque 9.1. Un index accélère les lectures mais ralentit les écritures (insertions, mises à jour, suppressions) car la structure doit être maintenue. Le compromis lecture/écriture guide le choix des index.

9.2 Arbres B et B$^+$

Définition 9.2 (Arbre B d'ordre m)**.** Un **arbre B** d'ordre m est un arbre de recherche équilibré où chaque nœud interne a au plus m fils et au moins $\lceil m/2 \rceil$ fils (sauf la racine). Les clés sont triées dans chaque nœud et les données sont stockées dans tous les nœuds.

Définition 9.3 (Arbre B$^+$)**.** Un **arbre B$^+$** est une variante où :

- les données ne sont stockées que dans les **feuilles**,
- les nœuds internes ne contiennent que des clés de routage,

- les feuilles sont chaînées en liste doublement chaînée pour les parcours séquentiels.

Proposition 9.4 (Complexité de l'arbre B^+)**.** Pour un arbre B^+ d'ordre m contenant n clés :

- Hauteur : $h = O(\log_m n)$.
- Recherche, insertion, suppression : $O(\log_m n)$ accès disque.
- Parcours séquentiel d'un intervalle de k clés : $O(\log_m n + k/B)$ où B est le nombre de clés par feuille.

Attention

L'arbre B^+ est la structure d'indexation par défaut de la plupart des SGBD relationnels (PostgreSQL, MySQL/InnoDB, Oracle, SQL Server). Ne pas confondre avec l'arbre binaire de recherche (BST) qui n'est pas adapté aux accès disque.

```
-- Creation d'un index B+ (implicite) en SQL
CREATE INDEX idx_nom ON etudiants (nom);

-- Index sur plusieurs colonnes (index composite)
CREATE INDEX idx_nom_prenom ON etudiants (nom, prenom);

-- Verifier l'utilisation de l'index
EXPLAIN ANALYZE SELECT * FROM etudiants WHERE nom = 'Dupont';
```

9.3 Index de hachage

Définition 9.5 (Index de hachage)**.** Un **index de hachage** utilise une fonction h : clé $\to \{0, 1, \ldots, B-1\}$ pour mapper directement une clé à un compartiment (*bucket*). La recherche par égalité est en $O(1)$ en moyenne.

Remarque 9.6. Les index de hachage ne supportent **pas** les requêtes de plage (`WHERE x BETWEEN a AND b`) ni le tri. Ils sont adaptés uniquement aux recherches par égalité exacte.

Définition 9.7 (Hachage extensible)**.** Le **hachage extensible** (*extendible hashing*) utilise un répertoire de pointeurs indexés par les d premiers bits du hachage. Lorsqu'un compartiment déborde, on double le répertoire (ou on divise le compartiment) sans réorganiser toute la table.

9.4 Index bitmap

Définition 9.8 (Index bitmap)**.** Un **index bitmap** associe à chaque valeur distincte v d'une colonne un vecteur de bits B_v de longueur n (nombre de lignes) : $B_v[i] = 1$ si la ligne i a la valeur v, 0 sinon.

Proposition 9.9 (Efficacité des opérations logiques)**.** Les requêtes combinant plusieurs prédicats se traduisent en opérations bit à bit rapides :

- `WHERE couleur = 'rouge' AND taille = 'L'` $\rightarrow B_{\text{rouge}} \wedge B_{\text{L}}$.
- `WHERE couleur = 'rouge' OR taille = 'L'` $\rightarrow B_{\text{rouge}} \vee B_{\text{L}}$.

Le comptage se fait par `POPCOUNT` en $O(n/w)$ où w est la taille du mot machine.

Remarque 9.10. Les index bitmap sont particulièrement efficaces pour les colonnes à **faible cardinalité** (sexe, statut, catégorie). Ils sont utilisés massivement dans les entrepôts de données (Oracle, Vertica).

9.5 Index spatiaux : R-tree

Définition 9.11 (R-tree)**.** Un **R-tree** est un arbre équilibré pour l'indexation de données spatiales multidimensionnelles. Chaque nœud contient un **rectangle englobant minimum** (MBR) qui englobe tous les objets de son sous-arbre.

Exemple 9.12. Recherche spatiale : « Trouver tous les restaurants dans un rayon de 500m » :

```
-- PostGIS (extension spatiale de PostgreSQL)
CREATE INDEX idx_geom ON restaurants USING GIST (geom);

SELECT nom, ST_Distance(geom, ST_MakePoint(2.35, 48.86)::geography) AS dist
FROM restaurants
WHERE ST_DWithin(geom, ST_MakePoint(2.35, 48.86)::geography, 500)
ORDER BY dist;
```

9.6 Index plein texte

Définition 9.13 (Index inversé)**.** Un **index inversé** (*inverted index*) associe à chaque terme (mot) la liste des documents (lignes) qui le contiennent, avec éventuellement la position et la fréquence.

Exemple 9.14.

```
-- PostgreSQL : index GIN pour la recherche plein texte
CREATE INDEX idx_fts ON articles USING GIN (to_tsvector('french', contenu));

SELECT titre
FROM articles
WHERE to_tsvector('french', contenu) @@ to_tsquery('french', 'bayesien &
↪  statistique');
```

9.7 Stratégies de sélection d'index

Définition 9.15 (Problème de sélection d'index)**.** Étant donné une charge de travail (ensemble de requêtes avec leurs fréquences) et un budget d'espace disque, choisir l'ensemble d'index qui minimise le coût total d'exécution.

Règles heuristiques :

1. Indexer les colonnes apparaissant dans les clauses `WHERE`, `JOIN` et `ORDER BY`.
2. Préférer les index composites couvrants (*covering index*) pour éviter l'accès à la table.
3. Éviter d'indexer les colonnes à très haute cardinalité modifiées fréquemment.
4. Utiliser `EXPLAIN ANALYZE` pour vérifier l'utilisation effective des index.

Exemple 9.16

```
-- Index couvrant : toutes les colonnes de la requete sont dans l'index
CREATE INDEX idx_covering ON commandes (client_id, date_commande)
    INCLUDE (montant);

-- La requete suivante peut etre satisfaite sans acceder a la table
SELECT date_commande, montant
FROM commandes
WHERE client_id = 42
ORDER BY date_commande DESC;
```

9.8 Formules clés

Formules clés

- **B$^+$-tree** : recherche en $O(\log_m n)$ accès disque, parcours de plage en $O(\log_m n + k/B)$
- **Hash** : recherche par égalité en $O(1)$, pas de requêtes de plage
- **Bitmap** : opérations logiques en $O(n/w)$, idéal pour faible cardinalité
- **R-tree** : recherche spatiale en $O(\log n)$ en moyenne
- **Compromis** : accélération lecture vs. surcoût écriture

9.9 Exercices

Exercice 9.1. Dessiner l'arbre B$^+$ d'ordre 4 obtenu après insertion séquentielle des clés $3, 7, 1, 14, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16$. Montrer l'état après la suppression de la clé 14.

Exercice 9.2. Une table `employes` a 10 millions de lignes. La colonne `departement` a 50 valeurs distinctes, la colonne `salaire` est continue. Quel type d'index recommandez-vous pour chacune ? Justifier.

Exercice 9.3. Écrire une requête SQL qui ne peut *pas* utiliser un index de hachage mais peut utiliser un index B$^+$. Expliquer pourquoi.

Exercice 9.4. On dispose d'un index bitmap sur la colonne `statut` (3 valeurs : `actif`, `inactif`, `suspendu`) et d'un index bitmap sur `region` (10 valeurs). Combien d'opérations `AND` bit à bit sont nécessaires pour répondre à `WHERE statut = 'actif' AND region IN ('IDF', 'PACA')` ?

Exercice 9.5. Utiliser `EXPLAIN ANALYZE` dans PostgreSQL pour comparer les performances d'une requête avec et sans index. Mesurer le nombre de pages lues (*buffers*) et le temps d'exécution.

CHAPITRE 9. INDEXATION ET STRUCTURES DE DONNÉES

Chapitre 10

ETL et Entrepôts de Données

Les bases de données transactionnelles sont optimisées pour les opérations du quotidien : insérer une commande, mettre à jour un stock, enregistrer un paiement. Mais quand un dirigeant veut analyser les tendances de vente sur cinq ans, croiser les données de trois départements et générer un tableau de bord, la base transactionnelle n'est pas l'outil adapté. C'est pour répondre à ce besoin que Bill Inmon et Ralph Kimball, dans les années 1990, ont formalisé la notion d'*entrepôt de données* (data warehouse) : un système séparé, organisé en schéma en étoile, optimisé pour les requêtes analytiques. Les pipelines *ETL* (Extract, Transform, Load) sont les tuyaux qui alimentent cet entrepôt en nettoyant et transformant les données brutes provenant de sources hétérogènes.

Idée centrale

Un entrepôt de données (*data warehouse*) centralise les données de l'entreprise dans un schéma optimisé pour l'analyse décisionnelle. Les pipelines ETL (Extract, Transform, Load) alimentent cet entrepôt en nettoyant et transformant les données provenant de sources hétérogènes.

10.1 Concepts fondamentaux

Définition 10.1 (Entrepôt de données)**.** Un **entrepôt de données** (data warehouse) est une base de données orientée *sujet*, *intégrée*, *non volatile* et *historisée*, conçue pour le support à la décision (Inmon, 1992).

Remarque 10.2. L'entrepôt de données s'oppose aux bases **OLTP** (*Online Transaction Processing*) qui servent les opérations quotidiennes. L'analyse décisionnelle relève de l'**OLAP** (*Online Analytical Processing*).

Caractéristique	OLTP	OLAP
Opérations	Lectures/écritures courtes	Requêtes analytiques longues
Utilisateurs	Nombreux, opérationnels	Peu, analystes
Données	Courantes, détaillées	Historiques, agrégées
Schéma	Normalisé (3NF)	Dénormalisé (étoile, flocon)

10.2 Modélisation dimensionnelle

Définition 10.3 (Schéma en étoile)**.** Un **schéma en étoile** organise les données autour d'une **table de faits** centrale (mesures quantitatives) entourée de **tables de dimension** (attributs descriptifs). Les dimensions sont reliées à la table de faits par des clés étrangères.

Exemple 10.4. Entrepôt de données pour les ventes d'un commerce :

```
-- Table de faits
CREATE TABLE fait_ventes (
    id_vente     SERIAL PRIMARY KEY,
    id_produit   INT REFERENCES dim_produit(id),
    id_client    INT REFERENCES dim_client(id),
    id_date      INT REFERENCES dim_date(id),
    id_magasin   INT REFERENCES dim_magasin(id),
    quantite     INT,
    montant      DECIMAL(10,2),
    cout         DECIMAL(10,2)
);

-- Table de dimension
CREATE TABLE dim_date (
    id      INT PRIMARY KEY,
    jour    DATE,
    mois    INT,
    trimestre INT,
    annee   INT,
    jour_semaine VARCHAR(10)
);
```

Définition 10.5 (Schéma en flocon de neige)**.** Le **schéma en flocon de neige** est une variante où les tables de dimension sont normalisées : chaque hiérarchie est extraite dans une table séparée (ex. : `dim_ville` → `dim_region` → `dim_pays`).

Attention

Le schéma en étoile est généralement préféré au flocon car :

- les jointures sont plus simples (une seule étape),
- les performances sont meilleures (moins de jointures),
- la redondance est acceptable en OLAP (espace disque bon marché).

10.3 Processus ETL

Définition 10.6 (ETL)**.** Le processus **ETL** (*Extract, Transform, Load*) comprend :

1. **Extraction** : récupérer les données des sources (bases OLTP, fichiers CSV, API, logs).

2. **Transformation** : nettoyer, normaliser, dédupliquer, agréger, calculer des indicateurs dérivés.
3. **Chargement** : insérer les données transformées dans l'entrepôt.

Exemple 10.7. Pipeline ETL en Python avec pandas :

```python
import pandas as pd
from sqlalchemy import create_engine

# 1. Extract
df_ventes = pd.read_csv("ventes_2025.csv")
df_clients = pd.read_sql("SELECT * FROM clients", engine_oltp)

# 2. Transform
df_ventes["date"] = pd.to_datetime(df_ventes["date"])
df_ventes = df_ventes.dropna(subset=["montant"])
df_ventes["montant"] = df_ventes["montant"].clip(lower=0)
df = df_ventes.merge(df_clients, on="client_id", how="left")
df["trimestre"] = df["date"].dt.quarter

# 3. Load
engine_dw = create_engine("postgresql://user:pass@host/warehouse")
df.to_sql("fait_ventes", engine_dw, if_exists="append", index=False)
```

Définition 10.8 (ELT)**.** La variante **ELT** (*Extract, Load, Transform*) charge d'abord les données brutes dans l'entrepôt puis effectue les transformations directement en SQL. Cette approche est populaire avec les entrepôts cloud (BigQuery, Snowflake, Redshift) grâce à leur puissance de calcul.

10.4 Dimensions à évolution lente (SCD)

Définition 10.9 (Slowly Changing Dimensions)**.** Les **SCD** gèrent les modifications des attributs dimensionnels :

- **Type 1** : écraser l'ancienne valeur (pas d'historique).
- **Type 2** : créer une nouvelle ligne avec dates de validité (`date_debut`, `date_fin`, `est_courant`).
- **Type 3** : ajouter une colonne pour l'ancienne valeur (historique limité).

Exemple 10.10. SCD Type 2 — un client déménage :

```sql
-- Avant : ligne courante
-- id=1, nom='Dupont', ville='Paris', date_debut='2020-01-01',
--       date_fin='9999-12-31', est_courant=TRUE

-- Apres demenagement :
UPDATE dim_client SET date_fin = '2025-06-15', est_courant = FALSE
```

```
WHERE id = 1 AND est_courant = TRUE;

INSERT INTO dim_client (id_naturel, nom, ville, date_debut, date_fin,
↪  est_courant)
VALUES (1, 'Dupont', 'Lyon', '2025-06-15', '9999-12-31', TRUE);
```

10.5 Opérations OLAP

Définition 10.11 (Opérations OLAP)**.** Les opérations OLAP classiques sur un cube multidimensionnel sont :

- **Roll-up** (agrégation) : monter dans la hiérarchie (jour → mois → année).
- **Drill-down** (désagrégation) : descendre dans la hiérarchie.
- **Slice** : fixer une dimension (ex. : `annee = 2025`).
- **Dice** : sélectionner un sous-cube multi-dimensionnel.
- **Pivot** : rotation des axes du cube.

```
-- Roll-up : ventes par mois -> ventes par trimestre
SELECT d.trimestre, d.annee, SUM(f.montant) AS total
FROM fait_ventes f
JOIN dim_date d ON f.id_date = d.id
GROUP BY d.trimestre, d.annee
ORDER BY d.annee, d.trimestre;

-- Slice + Dice avec CUBE
SELECT d.annee, p.categorie, m.region,
       SUM(f.montant) AS total
FROM fait_ventes f
JOIN dim_date d ON f.id_date = d.id
JOIN dim_produit p ON f.id_produit = p.id
JOIN dim_magasin m ON f.id_magasin = m.id
WHERE d.annee IN (2024, 2025)
GROUP BY CUBE (d.annee, p.categorie, m.region);
```

10.6 Vues matérialisées

Définition 10.12 (Vue matérialisée)**.** Une **vue matérialisée** est une vue dont le résultat est physiquement stocké sur disque. Elle accélère les requêtes analytiques répétitives au prix d'un espace de stockage supplémentaire et d'un coût de rafraîchissement.

```
-- Creation d'une vue materialisee
CREATE MATERIALIZED VIEW mv_ventes_mensuelles AS
SELECT d.annee, d.mois, p.categorie,
       SUM(f.montant) AS total, COUNT(*) AS nb_ventes
```

```
FROM fait_ventes f
JOIN dim_date d ON f.id_date = d.id
JOIN dim_produit p ON f.id_produit = p.id
GROUP BY d.annee, d.mois, p.categorie;

-- Rafraichissement
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_ventes_mensuelles;
```

10.7 Formules clés

Formules clés

- **ETL** : Extract → Transform → Load (vs. ELT : Extract → Load → Transform)
- **Étoile** : 1 table de faits + k tables de dimensions, jointures simples
- **OLAP** : Roll-up, Drill-down, Slice, Dice, Pivot
- **SCD Type 2** : historisation par `date_debut`/`date_fin`
- **Vue matérialisée** : compromis espace vs. temps de requête

10.8 Exercices

Exercice 10.1. Concevoir un schéma en étoile pour un système de réservation de vols aériens. Identifier la table de faits, les mesures et au moins quatre dimensions.

Exercice 10.2. Transformer le schéma en étoile de l'exercice précédent en schéma en flocon. Discuter les avantages et inconvénients.

Exercice 10.3. Écrire un pipeline ETL en Python qui : (a) extrait des données d'un fichier JSON, (b) nettoie les valeurs manquantes et normalise les dates, (c) charge le résultat dans une table PostgreSQL.

Exercice 10.4. Implémenter une dimension à évolution lente de type 2 pour la table `dim_produit`. Simuler un changement de prix et vérifier que l'historique est préservé.

Exercice 10.5. Écrire les requêtes SQL correspondant aux opérations OLAP suivantes sur le schéma de ventes : (a) roll-up des ventes du jour au trimestre, (b) slice sur l'année 2025, (c) dice sur les produits électroniques vendus en Île-de-France.

Chapitre 11

Bases de Données NoSQL

Idée centrale

Les bases NoSQL (*Not Only SQL*) abandonnent le modèle relationnel strict pour gagner en scalabilité horizontale, en flexibilité de schéma et en performances sur des cas d'usage spécifiques. Quatre grandes familles existent : clé-valeur, document, colonne et graphe.

11.1 Motivation et contexte

Les bases relationnelles atteignent leurs limites face à :

- des volumes massifs (pétaoctets),
- des données semi-structurées ou non structurées,
- des besoins de haute disponibilité et de distribution géographique,
- des schémas évolutifs (développement agile).

Remarque 11.1. NoSQL ne signifie pas « anti-SQL »mais « Not Only SQL ». Beaucoup de bases NoSQL proposent désormais un langage de requête déclaratif (CQL pour Cassandra, MQL pour MongoDB, Cypher pour Neo4j).

11.2 Théorème CAP

Théorème 11.2 (Théorème CAP (Brewer, 2000))**.** *Un système distribué ne peut garantir simultanément que deux des trois propriétés suivantes :*

1. ***Consistency*** *(cohérence) : toute lecture renvoie la dernière écriture.*
2. ***Availability*** *(disponibilité) : toute requête reçoit une réponse (non nécessairement la plus récente).*
3. ***Partition tolerance*** *(tolérance aux partitions) : le système fonctionne malgré des coupures réseau.*

Attention

En pratique, les partitions réseau sont inévitables dans un système distribué. Le vrai choix est donc entre **CP** (cohérence forte, ex. : HBase, MongoDB avec `readConcern: majority`) et **AP** (disponibilité, ex. : Cassandra, DynamoDB en mode éventuel).

Définition 11.3 (Cohérence éventuelle)**.** Un système à **cohérence éventuelle** (*eventual consistency*) garantit que, en l'absence de nouvelles écritures, toutes les répliques convergent vers la même valeur après un délai fini.

11.3 Bases clé-valeur

Définition 11.4 (Base clé-valeur)**.** Une **base clé-valeur** stocke des paires (k, v) où k est une clé unique et v une valeur opaque (chaîne, blob, objet sérialisé). L'accès se fait uniquement par clé : `GET`, `SET`, `DELETE`.

Exemple 11.5. Redis : cache en mémoire et structure de données :

```
import redis

r = redis.Redis(host='localhost', port=6379, db=0)
r.set('user:1001:name', 'Alice')
r.set('user:1001:email', 'alice@example.com')
r.expire('user:1001:name', 3600)  # TTL de 1 heure

name = r.get('user:1001:name')  # b'Alice'

# Structures avancees
r.hset('user:1002', mapping={'name': 'Bob', 'age': '25'})
r.lpush('queue:jobs', 'task_42')
r.zadd('leaderboard', {'Alice': 1500, 'Bob': 1200})
```

Remarque 11.6. Cas d'usage typiques : cache applicatif, sessions utilisateur, compteurs temps réel, files d'attente de messages.

11.4 Bases document

Définition 11.7 (Base document)**.** Une **base document** stocke des documents semi-structurés (JSON, BSON) regroupés en *collections*. Chaque document a un schéma flexible : deux documents de la même collection peuvent avoir des champs différents.

Exemple 11.8. MongoDB :

```
from pymongo import MongoClient

client = MongoClient('mongodb://localhost:27017/')
db = client['universite']
```

```
col = db['etudiants']

# Insertion
col.insert_one({
    'nom': 'Dupont',
    'prenom': 'Alice',
    'notes': [
        {'matiere': 'BDD', 'note': 16},
        {'matiere': 'Algo', 'note': 14}
    ],
    'adresse': {'ville': 'Paris', 'cp': '75005'}
})

# Requete avec filtre et projection
resultats = col.find(
    {'notes.note': {'$gte': 15}},
    {'nom': 1, 'prenom': 1, '_id': 0}
)

# Agregation (equivalent GROUP BY)
pipeline = [
    {'$unwind': '$notes'},
    {'$group': {'_id': '$nom', 'moyenne': {'$avg': '$notes.note'}}}
]
col.aggregate(pipeline)
```

Remarque 11.9. MongoDB offre des index secondaires, des transactions multi-documents (depuis la version 4.0) et un langage d'agrégation riche. Il convient aux données à schéma variable (catalogues produits, CMS, IoT).

11.5 Bases orientées colonnes

Définition 11.10 (Base orientée colonnes)**.** Une **base orientée colonnes** (ou *column-family store*) organise les données par familles de colonnes plutôt que par lignes. Chaque ligne est identifiée par une clé de partition et peut avoir un nombre variable de colonnes.

Exemple 11.11. Apache Cassandra :

```
-- CQL (Cassandra Query Language)
CREATE KEYSPACE universite WITH replication = {
    'class': 'SimpleStrategy', 'replication_factor': 3
};

CREATE TABLE universite.notes (
    etudiant_id UUID,
    matiere TEXT,
    semestre INT,
    note FLOAT,
    PRIMARY KEY ((etudiant_id), matiere, semestre)
) WITH CLUSTERING ORDER BY (matiere ASC, semestre DESC);
```

```
-- La cle de partition (etudiant_id) determine le noeud de stockage
-- Les colonnes de clustering ordonnent les donnees au sein de la partition
```

Remarque 11.12. Cassandra excelle pour les écritures massives distribuées (logs, séries temporelles, IoT). La modélisation est pilotée par les requêtes : on conçoit les tables en fonction des `SELECT` que l'on souhaite exécuter.

11.6 Bases de données orientées graphe

Définition 11.13 (Base de données graphe)**.** Une **base de données graphe** stocke des **nœuds** (entités) et des **arêtes** (relations) comme citoyens de première classe. Les traversées de graphe sont des opérations natives en $O(1)$ par relation (indépendamment de la taille du graphe).

Exemple 11.14. Neo4j et le langage Cypher :

```
// Creer des noeuds et des relations
CREATE (a:Personne {nom: 'Alice', age: 30})
CREATE (b:Personne {nom: 'Bob', age: 25})
CREATE (a)-[:AMIS_AVEC {depuis: 2020}]->(b)
CREATE (a)-[:TRAVAILLE_A]->(:Entreprise {nom: 'TechCorp'})

// Trouver les amis d'amis (distance 2)
MATCH (p:Personne {nom: 'Alice'})-[:AMIS_AVEC*2]-(fof)
WHERE fof <> p
RETURN DISTINCT fof.nom

// Plus court chemin
MATCH path = shortestPath(
    (a:Personne {nom: 'Alice'})-[*..6]-(b:Personne {nom: 'Charlie'})
)
RETURN path
```

Remarque 11.15. Les bases graphe sont idéales pour les réseaux sociaux, la détection de fraude, les systèmes de recommandation et la gestion des connaissances (knowledge graphs). Elles surpassent les bases relationnelles dès que les requêtes impliquent des jointures récursives profondes.

11.7 Comparaison et choix

	Clé-valeur	**Document**	**Colonne**	**Graphe**
Exemple	Redis	MongoDB	Cassandra	Neo4j
Schéma	Aucun	Flexible	Semi-fixe	Flexible
Requêtes	Par clé	Riches	Par partition	Traversées
Scalabilité	Très haute	Haute	Très haute	Modérée
Cas typique	Cache	CMS, catalogue	Séries temp.	Réseau social

Proposition 11.16 (Critères de choix)**.** Le choix d'un type de base NoSQL dépend de :

1. la **structure** des données (schéma fixe vs. flexible),
2. les **patterns d'accès** (clé, requêtes complexes, traversées),
3. les exigences de **cohérence** vs. **disponibilité**,
4. le **volume** et la **vélocité** des données.

11.8 Formules clés

Formules clés

- **CAP** : Consistency + Availability + Partition tolerance — au plus 2 sur 3
- **Clé-valeur** : accès $O(1)$ par clé, aucune requête sur la valeur
- **Document** : schéma flexible, index secondaires, agrégation
- **Colonne** : modélisation par requête, écritures distribuées massives
- **Graphe** : traversées en $O(k)$ où k = nombre d'arêtes traversées

11.9 Exercices

Exercice 11.1. Pour chacun des cas d'usage suivants, recommander un type de base NoSQL et justifier : (a) cache de sessions web, (b) catalogue de produits avec attributs variables, (c) suivi de capteurs IoT à 100 000 événements/s, (d) détection de communautés dans un réseau social.

Exercice 11.2. Écrire un script Python qui insère 10 000 documents dans MongoDB, crée un index sur un champ, puis compare le temps d'une requête avec et sans index.

Exercice 11.3. Modéliser un système de recommandation de films dans Neo4j. Créer les nœuds (utilisateurs, films, genres) et les relations (A_VU, A_NOTE, EST_DU_GENRE). Écrire une requête Cypher pour recommander des films à un utilisateur basé sur les préférences de ses amis.

Exercice 11.4. Concevoir le schéma Cassandra pour une application de messagerie (messages entre utilisateurs). La requête principale est « afficher les N derniers messages d'une conversation ». Justifier le choix de la clé de partition et des colonnes de clustering.

Exercice 11.5. Discuter les implications du théorème CAP pour une application bancaire distribuée sur plusieurs continents. Quel compromis recommanderiez-vous ? Pourquoi ?

Bibliographie

[1] Ramakrishnan, R. and Gehrke, J., *Database Management Systems*, 3rd ed., McGraw-Hill, 2003.

[2] Date, C.J., *An Introduction to Database Systems*, 8th ed., Pearson, 2004.

[3] Silberschatz, A., Korth, H.F. and Sudarshan, S., *Database System Concepts*, 7th ed., McGraw-Hill, 2020.