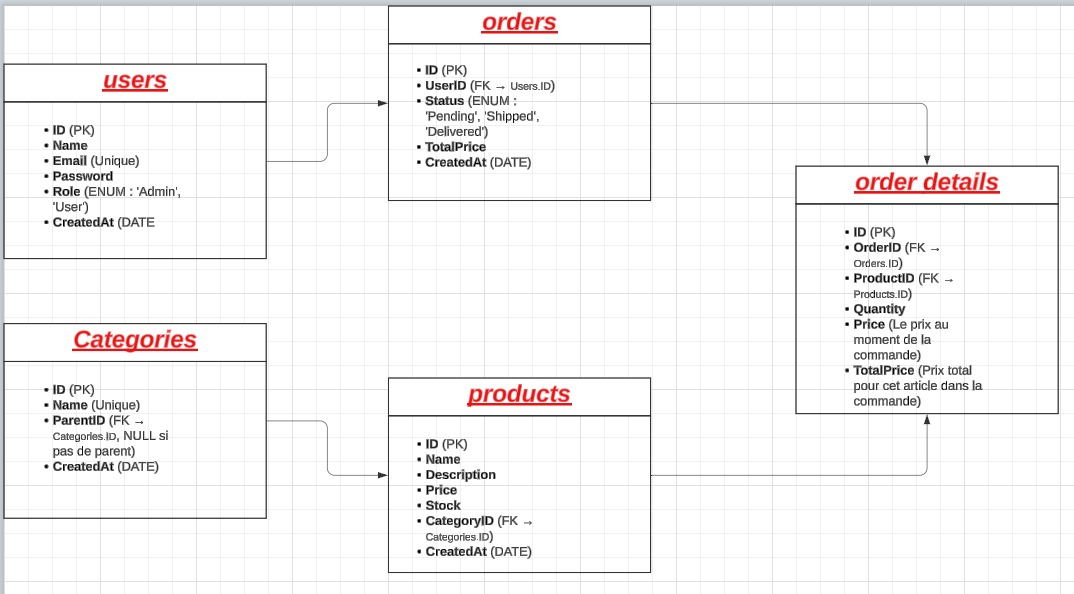
**1. Analyse des besoins détaillée**

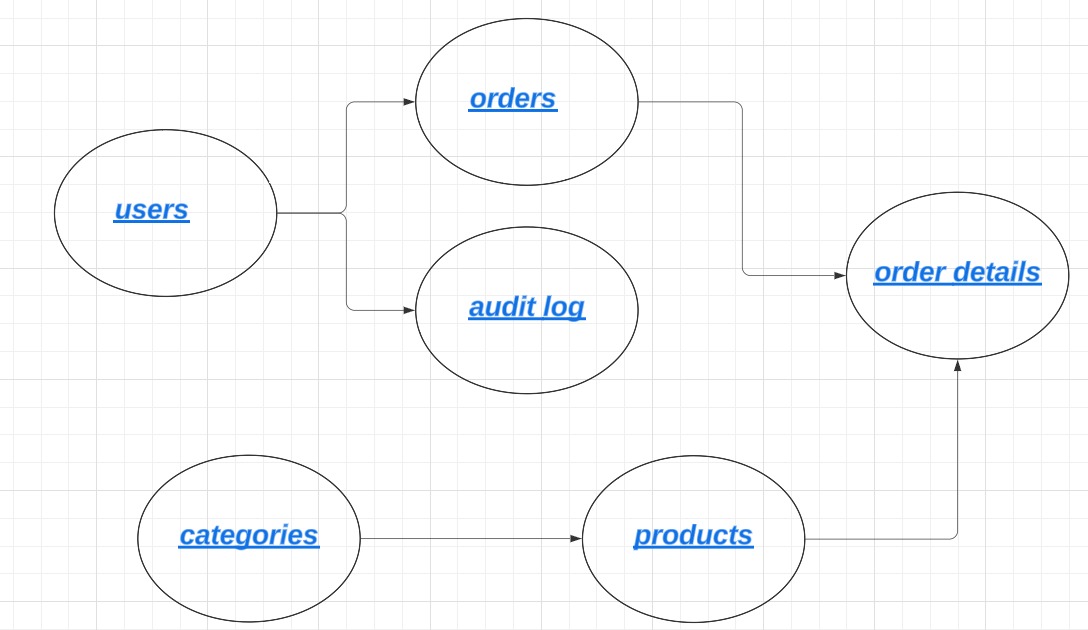
Le projet consiste en la mise en place d’un système de gestion pour une plateforme de commerce électronique. Le but est de répondre aux besoins suivants :

* **Gérer les utilisateurs :** Le système doit permettre l'enregistrement des utilisateurs, en leur attribuant des rôles spécifiques (par exemple, Admin et Utilisateur). L'admin aura accès à des fonctionnalités avancées comme la gestion des produits et des commandes, tandis que les utilisateurs auront un accès limité à la navigation et à l'achat de produits.
* **Organiser les catégories** : L’organisation des produits doit être structurée selon une hiérarchie de catégories. Ce système permet de classer les produits dans des catégories et sous-catégories afin de faciliter la navigation des utilisateurs et de permettre une gestion simplifiée du catalogue de produits.
* **Gérer les produits** : Le système doit permettre la gestion complète des produits, y compris la mise à jour de leurs informations, le suivi des stocks et la gestion des prix.
* **Suivre les commandes** : Le suivi des commandes est essentiel pour assurer une gestion fluide et une bonne expérience utilisateur. Le système doit permettre de gérer toutes les étapes d’une commande, de la création à la livraison.
* **Auditer les modifications** : Pour assurer la traçabilité et la sécurité du système, toutes les modifications importantes doivent être enregistrées dans une table d’audit. Cela permet de suivre qui a effectué chaque changement, quand et pourquoi.

# 2. **Diagramme entité-relation complet**

Le schéma ER montre : ****

# 3. Schéma de la base de données normalisée



# 4.Justification des choix techniques

**1.Structure des Tables**

Les tables principales comprennent Users, Categories, Products, Orders, OrderDetails, et audit\_log. Chacune de ces tables a été pensée pour :

* **Modélisation logique des entités** : Chaque table représente une entité distincte du système (utilisateur, produit, commande, etc.), avec des clés primaires pour garantir l'unicité des enregistrements.
* **Références entre les entités** : Les tables sont reliées par des clés étrangères, ce qui garantit l'intégrité référentielle entre les entités (par exemple, un OrderDetails référence un produit via ProductID et une commande via OrderID).
* **Gestion des relations hiérarchiques** : La table Categories inclut un champ ParentID pour permettre une hiérarchisation des catégories de produits. Cela permet de gérer des structures de catégories complexes, comme des sous-catégories, tout en maintenant la flexibilité.

**Choix des types de données**

* **VARCHAR pour les chaînes de caractères** : Le type VARCHAR est utilisé pour des champs comme Name, Email, et Description des produits et des utilisateurs. Il est plus flexible que CHAR car il ne consomme de l'espace que pour la taille réelle de la chaîne.
* **ENUM pour les rôles et états** : Les types ENUM (par exemple, pour Role dans Users ou Status dans Orders) sont utilisés pour garantir que seules des valeurs spécifiques peuvent être insérées dans la table. Cela permet de simplifier la gestion des états et des rôles tout en optimisant les requêtes.
* **DECIMAL pour les prix** : Le type DECIMAL(10,2) est utilisé pour les champs relatifs aux prix afin de garantir une précision exacte pour les calculs financiers (par exemple, Price dans Products et TotalPrice dans Orders).
* **JSON pour l'audit log** : Le type JSON est utilisé pour stocker les anciennes et nouvelles données dans la table audit\_log, ce qui permet une grande flexibilité pour enregistrer les modifications sans avoir à ajouter des colonnes pour chaque champ modifié.

**2. Types d'Index**

Les index sont essentiels pour améliorer les performances des requêtes, en particulier lorsque la base de données se développe et que le nombre de lignes augmente.

**Index primaire (Primary Key)**

* **Utilisation** : Chaque table dispose d'une clé primaire (par exemple, ID dans Users, Products, etc.). La clé primaire est un index unique sur la table et permet d'assurer que chaque enregistrement peut être récupéré de manière rapide et sans ambiguïté.
* **Justification** : Les clés primaires sont utilisées pour identifier de manière unique chaque enregistrement et sont essentielles pour la gestion des relations entre les tables via les clés étrangères.

**Index sur les clés étrangères**

* **Utilisation** : Des index sont automatiquement créés sur les colonnes référencées par des clés étrangères (par exemple, CategoryID dans Products, UserID dans Orders, et ProductID et OrderID dans OrderDetails).
* **Justification** : Ces index permettent de rendre les jointures plus efficaces. Par exemple, lorsqu'on joint Orders et Users sur UserID, un index sur UserID permet de rendre cette opération beaucoup plus rapide.

**Index sur les colonnes fréquemment recherchées**

* **Exemple** : Création d'index sur les colonnes comme Email dans la table Users et Name dans la table Products.
* **Justification** : Ces index accélèrent les recherches, les filtres et les tris, car ces colonnes sont souvent utilisées dans les requêtes de recherche ou de filtrage.

**Type d'index : B-tree vs Hash**

* **B-tree** : La plupart des index créés (par exemple, sur les clés primaires et étrangères) sont de type B-tree. Les index B-tree sont efficaces pour les requêtes qui impliquent des plages de valeurs (par exemple, rechercher tous les produits dont le prix est supérieur à une certaine valeur) et pour les opérations de tri.
* **Hash** : Les index de type hash peuvent être utilisés pour des recherches très spécifiques où l'on recherche exactement une valeur (par exemple, rechercher un utilisateur par email). Cependant, dans une base relationnelle, les index B-tree sont généralement préférés, car ils permettent une gestion plus souple des requêtes.

**3. Stratégie de Partitionnement**

Le partitionnement de la base de données est une technique permettant de diviser une grande table en plusieurs sous-ensembles appelés partitions. Cela peut améliorer les performances, la gestion de la maintenance et la scalabilité des bases de données très volumineuses.

**Partitionnement des données**

* **Partitionnement horizontal** : Cela consiste à diviser une grande table en plusieurs partitions basées sur un critère de ligne, tel qu'une plage de dates ou un identifiant. Par exemple :
  + **Table Orders** : Le partitionnement peut être effectué en fonction de la date de création des commandes (CreatedAt). Ainsi, les commandes anciennes peuvent être stockées dans une partition différente des commandes récentes, ce qui améliore les performances des requêtes concernant des commandes récentes.

CREATE TABLE Orders (

ID INT AUTO\_INCREMENT PRIMARY KEY,

UserID INT NOT NULL,

Status ENUM('Pending', 'Shipped', 'Delivered') NOT NULL DEFAULT 'Pending',

TotalPrice DECIMAL(10,2) NOT NULL,

CreatedAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

PARTITION BY RANGE (YEAR(CreatedAt)) (

PARTITION p2023 VALUES LESS THAN (2024),

PARTITION p2024 VALUES LESS THAN (2025)

);

* **Partitionnement basé sur la clé primaire** : Dans des cas où il est difficile de déterminer un critère de partitionnement, le partitionnement peut être basé sur la clé primaire (ID). Chaque partition pourrait ainsi contenir un sous-ensemble d'ID.

**Partitionnement des tables de journalisation (audit\_log)**

* **Objectif** : La table audit\_log peut devenir très volumineuse, car chaque mise à jour ou suppression entraîne l’insertion d'un enregistrement. Le partitionnement de cette table peut être effectué en fonction de la date (timestamp) afin de diviser les logs par mois ou par année.

CREATE TABLE audit\_log (

id INT AUTO\_INCREMENT PRIMARY KEY,

table\_name VARCHAR(255),

operation VARCHAR(10),

old\_data JSON,

new\_data JSON,

timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP

)

PARTITION BY RANGE (YEAR(timestamp)) (

PARTITION p2023 VALUES LESS THAN (2024),

PARTITION p2024 VALUES LESS THAN (2025)

);

# 5.Documentation des procédures stockées et triggers

**a) Procédures stockées**

**Exemples de procédures créées :**

* 1. **Ajout :**
  2. CREATE PROCEDURE create\_category(IN p\_name VARCHAR(255), IN p\_parent\_id INT) BEGIN INSERT INTO Categories (Name, ParentID) VALUES (p\_name, p\_parent\_id); END;

#### **b) Triggers**

Des triggers sont utilisés pour suivre les modifications . Exemple d'audit lors d'une mise à jour des utilisateurs :

CREATE TRIGGER log\_update\_users

AFTER UPDATE ON Users

FOR EACH ROW

BEGIN

INSERT INTO audit\_log (table\_name, operation, old\_data, new\_data)

VALUES ('Users', 'UPDATE',

JSON\_OBJECT('id', OLD.ID, 'name', OLD.Name, 'email', OLD.Email, 'role', OLD.Role, 'created\_at', OLD.CreatedAt),

JSON\_OBJECT('id', NEW.ID, 'name', NEW.Name, 'email', NEW.Email, 'role', NEW.Role, 'created\_at', NEW.CreatedAt));

END;

**Exemple de suppression des commandes** :

CREATE TRIGGER log\_delete\_orders BEFORE DELETE ON Orders FOR EACH ROW BEGIN INSERT INTO audit\_log (table\_name, operation, old\_data) VALUES ('Orders', 'DELETE', JSON\_OBJECT('id', OLD.ID, 'user\_id', OLD.UserID, 'status', OLD.Status, 'total\_price', OLD.TotalPrice, 'created\_at', OLD.CreatedAt)); END;