# Gestion des données
# Compléments

HEI 2021 / 2022

# Retour sur la valeur NULL

- Signification
  - Valeur manquante ou inconnue pour l'instant
  - Non applicable : pas de valeur pour le cas donné
- Valeur non autorisée pour les clés primaires
- Comportement particulier

# Retour sur la valeur NULL

- Comportement particulier
  - Test de valeur
  - Contrainte unique

# Retour sur la valeur NULL

- Comportement particulier
  - Test de valeur

```
SELECT * FROM client WHERE solde=NULL;
```
← ne retourne rien

Utilisation du mot clé IS

```
SELECT * FROM client WHERE solde IS NULL;
```
← Ok, valeurs trouvées

  - Contrainte unique

# Retour sur la valeur NULL

- Comportement particulier
  - Test de valeur
  - Contrainte unique

```
CREATE TABLE test (id INT, valeur VARCHAR(30) UNIQUE);
INSERT INTO test(id) VALUES (1);
INSERT INTO test(id) VALUES (2);
SELECT * from test;
```

← Insertion ok

← Insertion ok aussi !!!

| id | valeur |
|---|---|
| 1 | *NULL* |
| 2 | *NULL* |

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

Récupérer la personne avec le plus grand solde ?

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

Récupérer la personne avec le plus grand solde ?
3 requêtes possibles :

```
1. SELECT c1.prenom FROM client c1 WHERE NOT
   EXISTS (SELECT * FROM client c2 WHERE
   c2.solde > c1.solde);
2. SELECT prenom FROM client WHERE solde =
   (SELECT max(solde) FROM client);
3. SELECT prenom FROM client WHERE solde >=ALL
   (SELECT solde FROM client);
```

EXISTS permet de sélectionner une valeur si la sous requête retourne un résultat
ALL permet de comparer une valeur avec l'ensemble des valeurs d'une sous requête

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

Récupérer la personne avec le plus grand solde ?

3 requêtes possibles :

1. `SELECT c1.prenom FROM client c1 WHERE NOT EXISTS (SELECT * FROM client c2 WHERE c2.solde > c1.solde);`

   → retourne « Amandine, Capucine »
2. `SELECT prenom FROM client WHERE solde = (SELECT max(solde) FROM client);`

   → retourne Amandine
3. `SELECT prenom FROM client WHERE solde >=ALL (SELECT solde FROM client);`

   → ne retourne rien

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

Récupérer la personne avec le plus grand solde ?
3 requêtes possibles :

```
1.  SELECT c1.prenom FROM client c1 WHERE NOT
    EXISTS (SELECT * FROM client c2 WHERE
    c2.solde > c1.solde);
```

→ retourne « Amandine, Capucine »

La comparaison d'un solde avec NULL sera évalué à false, du coup le not exist de () retourne true et Capucine est donc sélectionnée.

# Retour sur la valeur NULL

- Comparaison
- Table client

| id | nom | prenom | Solde |
|---|---|---|---|
| 1 | Dupont | Amandine | 200 |
| 2 | Durant | Adrien | 20 |
| 5 | Chandonnet | Fred | 120 |
| 8 | Saucier | Capucine | NULL |
| 15 | Dostie | Laurent | 150 |

Récupérer la personne avec le plus grand solde ?

3 requêtes possibles :

1. 
```
SELECT prenom FROM client WHERE solde >=ALL
(SELECT solde FROM client);
```
→ ne retourne rien

La condition ALL est vraie si la comparaison avec toutes les autres valeur est vraie, ici la comparaison avec NULL retourne false donc la comparaison globale retourne false

# Index

- Index : structure de donnée qui permet d'accélérer l'accès aux données
- Technique
  - Table de hachage
  - Arbre de recherche (BTREE)
- Avantage
  - Requêtes portant sur une colonne indexée plus rapides
- Inconvénient
  - Nécessite de recalculer l'index à chaque insertion dans la table
  - Stockage de l'index

# Index

- Index implicites
  - Clé primaire
  - Colonne « unique »
- Index créés
  - Syntaxe :

```
CREATE INDEX nomIndex ON table (col1 [ASC|DESC], col2, …);
DROP INDEX nomIndex;
```

# Index

- Sur quelle colonne créer un index ?
  - Colonnes utilisées comme critère de jointure
  - Colonnes utilisées comme critère de sélection
- Prendre en compte l'utilisation de la table
  - Rapport lecture/écriture

# Index

- Utilisation de l’index :
  - SELECT * FROM client WHERE id = 500;
  - SELECT * FROM client WHERE id > 10;
  - SELECT * FROM client WHERE id BETWEEN 300 AND 500;
  - SELECT * FROM client WHERE nom ='Martin’;
  - SELECT * FROM client WHERE nom LIKE 'M%' ;
- Index non-utilisé:
  - SELECT * FROM client;
  - SELECT * FROM client WHERE nom IS NULL;
  - SELECT * FROM client WHERE ca*10 >10000 ;

# Gestion des droits

- Les SGBD permettent à plusieurs utilisateurs d'accéder à la base
  - Pas les même besoins selon l'appli (admin vs client)
- Création d'un user

```
CREATE USER 'test' IDENTIFIED BY 'pwd';
```

- Ajout des droits pour un user

```
GRANT privilege ON table TO user [WITH GRANT OPTION];
```

- Suppression d'un droit

```
REVOKE privilege ON table FROM user;
```

- Lister les droits

```
SHOW GRANTS FOR user;
```

# Gestion des droits

- Privilèges :
  - SELECT (lecture)
  - INSERT (insertion)
  - UPDATE [(col1, col2,…)] (modification, peut être contraint à certaines colonnes)
  - DELETE (suppression)
  - ALTER (modification structure de la table)
  - ALL (tous les droits au dessus)
  - GRANT OPTION : (ajout de droits)

# Gestion des droits

- Exemple:

```
CREATE USER 'readUser' IDENTIFIED BY 'pwdRe@d';
GRANT SELECT ON tp01.fournisseur TO 'readUser';
GRANT SELECT(id, nom) ON tp01.client TO 'readUser';

CREATE USER 'writeUser' IDENTIFIED BY 'pwdWr1te';
GRANT SELECT,INSERT,UPDATE,DELETE ON tp01.* TO 'writeUser';

CREATE USER 'adminUser' IDENTIFIED BY 'adm1n';
GRANT ALL ON tp01.* TO 'adminUser';
```

# Gestion des droits

- Plusieurs appli (users) accèdent à l'appli avec les même besoin
  - Partager le user/mot de passe ?
  - Créer plusieurs user avec les même droits ?

# Gestion des droits

- Plusieurs appli (users) accèdent à l'appli avec les même besoin
  - Partager le user/mot de passe ?
  - Créer plusieurs user avec les même droits ?
  - ➢ Utilisation de rôles
    - Regroupe un ensemble de droits
    - Un rôle peut être assigné à plusieurs utilisateurs

# Gestion des droits

- Syntaxe

```
CREATE ROLE 'nomRole1', 'nomRole2',…;
GRANT privilege TO 'nomRole';
GRANT 'nomRole' TO 'user';
```

# Gestion des droits

- Exemple:

```
CREATE ROLE 'read', 'write', 'admin';

GRANT SELECT ON tp01.* TO 'read';
GRANT INSERT,UPDATE,DELETE ON tp01.* TO 'write';
GRANT ALL ON tp01.* TO 'admin';

GRANT 'read' to readUser;
GRANT 'write' to writeUser;
GRANT 'admin' to adminUser;
```

# Les Triggers

- Base de données actives
- Procédure associé à un évènement déclencheur
  - Evènement (INSERT, UPDATE, DELETE)
- Exécution automatique chaque fois que l'évènement se produit
  - Instant (BEFORE, AFTER)

# Les Triggers

- Syntaxe :

```
CREATE|REPLACE TRIGGER nomTrigger
AFTER|BEFORE { INSERT [ OR DELETE [OR UPDATE ON nomTable]]]}
[FOR EACH ROW]
actions;
```

- Suppression :

```
DROP TRIGGER nom;
```

# Les Triggers

- Exemple :

S’assurer que le nom des produits insérés soient en majuscule

```
DELIMITER | -- On change le délimiteur de fin de ligne
CREATE TRIGGER tp01.produitMaj
BEFORE INSERT ON tp01.produit
FOR EACH ROW -- Pour chaque ligne insérée
BEGIN
SET NEW.nom = UPPER(NEW.nom); -- mot clé NEW pour accéder à la valeur mise à jour
END |
DELIMITER ; -- On remet le délimiteur de fin de ligne par défaut
```

- Ne pas oublier de créer le même trigger pour la commande update

# Les Triggers

- OLD et NEW permettent d’accéder aux valeurs concernées avant (OLD) ou après (NEW) que la valeur ne soit modifiée par la requête qui a déclenché le trigger.

# Les Procédures stockées

- Série d'instructions SQL stockée en bdd
- Fonction qui peut être appelée en sql
- Avantage :
  - Permet de regrouper plusieurs requêtes (évite les A/R client-serveur)
  - Sécurité des données (UPDATE et DELETE gérés côté bdd)
- Inconvénient :
  - Ajoute traitement et stockage côté bdd
  - Syntaxe par rapport à un langage de développement classique

# Les Procédures stockées

- Syntaxe

```
DELIMITER | -- remplace le caractère de fin de ligne par |
CREATE [OR REPLACE] PROCEDURE nomProcedure(param1 [IN|OUT|INOUT] type, param2
type,…)
actions;
DELIMITER ; -- on remet le ;
```

- Appel d'une procédure :

```
CALL nomProcedure(param1, param2,…);
```

- Suppression :
- `DROP PROCEDURE nomProcedure;`

# Les Procédures stockées

- Exemples

```
DELIMITER |
CREATE procedure tp01.ajouterCommande(idClient INT, idProduit INT, qte INT)
BEGIN
  DECLARE idC INT;
  DECLARE idP INT;
  SELECT id INTO idC FROM client where id=idClient;
  SELECT id INTO idP FROM produit where id=idProduit;
  IF idC IS NOT NULL AND idP IS NOT NULL
    THEN
     INSERT INTO commande VALUES (idClient, idProduit, qte);
    ELSE
      SELECT 'Erreur';
    END IF;
END|
DELIMITER ;

CALL ajouterCommande(1,3,5);
```

# Les Procédures stockées

- Exemples

```
DELIMITER |
Create procedure tp01.calculerGain(idFournisseur INT, OUT prix INT)
BEGIN
  SELECT sum(p.prix_unitaire * c.quantite)INTO prix
 FROM fournisseur f
 JOIN produit p ON p.fournisseur = f.id
 JOIN commande c ON p.id = c.idProduit
 WHERE f.id=idFournisseur;
END|
DELIMITER ;

CALL calculerGain(1, @gain);
SELECT @gain;
```

# Les transactions

- L’état d’une bdd doit toujours être cohérent
  - Résistances aux pannes (logicielle/matérielle)
  - Gestion des accès concurrents à la base
- Préserver l’intégrité des données

# Les transactions

- Une transaction est une suite de modifications ordonnées dans la base qui forme une action unique (INSERT, DELETE, UPDATE)
- Concept ACID
  - Atomicité : une transaction d'effectue entièrement ou pas du tout
  - Consistance : une transaction  doit conserver la base dans un état cohérent
  - Isolation : pas d'interférence avec les autres utilisateurs
  - Durabilité : les actions d'une transaction terminée sont prisé en compte par la bdd

# Les transactions

- Atomicité : Une transaction se termine :
  - Soit à la validation de celle-ci avec l'ordre COMMIT;
  - Soit en cas d'erreur avec l'ordre ROLLBACK;
- Consistance :
  - Si une panne se produit pendant la validation ou l'annulation d'une transaction, le SGBD essaie de terminer la transaction ou remet la BDD dans son état avant la transaction

# Les transactions

- Isolation
  - Lors d'un SELECT le SGBD affiche:
    - Les données déjà validées lors de précédentes transactions
    - Les données créées ou mises à jour dans la transaction courante
    - Les données **détruites** dans d'autres transaction en cours

# Les transactions

- Isolation
  - Lors d'un SELECT le SGBD n'affiche pas :
    - Les données supprimées lors de précédentes transactions
    - Les données supprimées dans la transaction courante
    - Les données **créées ou mises à jour** dans d'autres transaction en cours

# Les transactions

- Problème : Comment gérer la mise à jour d'une valeur modifiée par 2 transactions ?
- Tant que la transaction n'est pas terminée le SGBD ne sait pas si celle-ci est validée ou rollbackée

# Exemple

Transaction #1
x=read(A)
x=x+10
write(A,x)

Transaction #2
y=read(A)
y=y+20
Write(A,y)

Transaction séquentielles :
A = 50
T1 puis T2 ou T2 puis T1;
A=80

Transaction concurrentes :

| T1 | T2 |
|---|---|
| x=read(A) | |
| x=x+10 | |
| | y=read(A) |
| write(A,x) | |
| | y=y+20 |
| | write(A,y) |

Après exécution A=70
→perte de l'action T1

# Exemple

Contrainte de la bdd : A=B

| T1 | T2 |
| --- | --- |
| x=read(A) | |
| x=x+1 | |
| write(A,x) | |
| | z=read(A) |
| | z=z*2 |
| | write(A,z) |
| | t=read(B) |
| | t=t*2 |
| | write(B,t) |
| y=read(B) | |
| y=y+1 | |
| write(B,y) | |



Transaction séquentielles :
A=B 50
T1 puis T2 : A=B=102
T2 puis T1 : A=B=101

Exécution concurrentes :
A = 102, B=101

# Les transactions

- Exécution correcte :
  - Le contrôleur de concurrence doit garantir que l'exécution simultanée de transactions produit le même résultat qu'une exécution séquentielle
  - Utilisation de verrous

# Les transactions

- Chaque mise à jour d'une ligne d'une table provoque le verrouillage de la ligne
- Le verrou est annulé à la fin de la transaction qui a posé le verrou (lock)
- Chaque mise à jour d'une ligne vérrouillée provoque le blocage de la transaction
- Le blocage de la transaction est annulé lorsque la transaction qui avait posé le verrou se termine

# Les transactions

- Problème interblocage (deadlocks)
  - T1 met à jour L1 : verrouillage de L1
  - T2 met à jour L2 : verrouillage de L2
  - T1 met à jour L2 : T1 est bloquée (attente de la libération de L2)
  - T2 met à jour L1 : T2 est bloquée (attente de la libération de L1)

  - →Les 2 transactions sont bloquées!!!

# Les transactions

- Problème interblocage (deadlocks)
  - T1 met à jour L1 : verrouillage de L1
  - T2 met à jour L2 : verrouillage de L2
  - T1 met à jour L2 : T1 est bloquée
  - T2 met à jour L1 :
    - T2 est bloquée, on annule la mise à jour de L2 par T1, T1 est débloquée, puis T2

→Perte d'une opération de mise à jour