Intégrité des données

Génie Biologique et Alimentaire 4ème année

Olivier Caron[1]
http://ocaron.polytech-lille.net

[1]École d'ingénieurs Polytech Lille
Université de Lille

15 novembre 2024

Contenu

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)
  - limiter l'accès des activités d'administration du serveur à un nombre minimal de machines du réseau.

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)
  - limiter l'accès des activités d'administration du serveur à un nombre minimal de machines du réseau.
  - auditer régulièrement les échecs de connexions via les logs, mise en place d'alerte

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)
  - limiter l'accès des activités d'administration du serveur à un nombre minimal de machines du réseau.
  - auditer régulièrement les échecs de connexions via les logs, mise en place d'alerte
  - mettre à jour régulièrement le SGBD

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)
  - limiter l'accès des activités d'administration du serveur à un nombre minimal de machines du réseau.
  - auditer régulièrement les échecs de connexions via les logs, mise en place d'alerte
  - mettre à jour régulièrement le SGBD
  - effectuer une veille technologique sur les techniques de **hacking**.

Gestion des utilisateurs, bonnes pratiques

- Chaque SGBD a sa propre base d'utilisateurs, l'activité du **DBA** (DataBase Administrator) consiste à :
  - maintenir régulièrement les utilisateurs référencés (suivi du personnel)
  - définir un niveau d'exigence des mots de passe (durée de vie du mot de passe, complexité du mot de passe)
  - limiter l'accès des activités d'administration du serveur à un nombre minimal de machines du réseau.
  - auditer régulièrement les échecs de connexions via les logs, mise en place d'alerte
  - mettre à jour régulièrement le SGBD
  - effectuer une veille technologique sur les techniques de **hacking**.
  - informer, documenter les développeurs

Injection SQL

Définition

L'injection SQL directe est une technique où un pirate modifie une requête SQL existante pour afficher des données cachées, ou pour écraser des valeurs importantes, ou encore exécuter des commandes dangereuses pour la base.

Injection SQL

Définition

L'injection SQL directe est une technique où un pirate modifie une requête SQL existante pour afficher des données cachées, ou pour écraser des valeurs importantes, ou encore exécuter des commandes dangereuses pour la base.

- L'injection SQL s'applique à n'importe quel langage de programmation (PHP, Java, ...),

Injection SQL

Définition

L'injection SQL directe est une technique où un pirate modifie une requête SQL existante pour afficher des données cachées, ou pour écraser des valeurs importantes, ou encore exécuter des commandes dangereuses pour la base.

- L'injection SQL s'applique à n'importe quel langage de programmation (PHP, Java, ...),
- De nombreuses variantes sur le détournement de `select`, `update`, ...

Injection SQL

Définition

L'injection SQL directe est une technique où un pirate modifie une requête SQL existante pour afficher des données cachées, ou pour écraser des valeurs importantes, ou encore exécuter des commandes dangereuses pour la base.

- L'injection SQL s'applique à n'importe quel langage de programmation (PHP, Java, ...),
- De nombreuses variantes sur le détournement de `select`, `update`, ...
- Importance de vérifier les données transmises !

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web
- La requête SQL va être construite par le code python suivant :

```
req=f"select 1 from user where name='{login}' and password='{password}'";
```

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web
- La requête SQL va être construite par le code python suivant :

```
req=f"select 1 from user where name='{login}' and password='{password}'";
```

- Si pas de ligne en résultat, l'authentification est refusée.

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web
- La requête SQL va être construite par le code python suivant :

```
req=f"select 1 from user where name='{login}' and password='{password}'";
```

- Si pas de ligne en résultat, l'authentification est refusée.
- Le hackeur saisit les chaînes "' or '1'='1" dans les champs `login` et `password`.

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web
- La requête SQL va être construite par le code python suivant :

```
req=f"select 1 from user where name='{login}' and password='{password}'";
```

- Si pas de ligne en résultat, l'authentification est refusée.
- Le hackeur saisit les chaînes "' or '1'='1" dans les champs `login` et `password`.
- La requête SQL générée sera donc :

```
"select 1 from user where name='' or '1'='1' and password='' or '1'='1'"
```

Exemple d'injection SQL

- Soit un programme qui vérifie le login/password saisi dans un formulaire web
- La requête SQL va être construite par le code python suivant :

```
req=f"select 1 from user where name='{login}' and password='{password}'";
```

- Si pas de ligne en résultat, l'authentification est refusée.
- Le hackeur saisit les chaînes "' or '1'='1" dans les champs `login` et `password`.
- La requête SQL générée sera donc :

```
"select 1 from user where name='' or '1'='1' and password='' or '1'='1'"
```

- Conséquence : l'authentification est acceptée !

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

  trust les connexions sont autorisées sans condition.

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

  trust les connexions sont autorisées sans condition.

  reject les connexions sont refusées sans condition.

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

trust	les connexions sont autorisées sans condition.
reject	les connexions sont refusées sans condition.
password	mot de passe transmis en clair.

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

trust	les connexions sont autorisées sans condition.
reject	les connexions sont refusées sans condition.
password	mot de passe transmis en clair.
crypt	mot de passe transmis en chiffré

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

trust	les connexions sont autorisées sans condition.
reject	les connexions sont refusées sans condition.
password	mot de passe transmis en clair.
crypt	mot de passe transmis en chiffré

- Connexions distantes, mode supplémentaire :

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

  trust les connexions sont autorisées sans condition.

  reject les connexions sont refusées sans condition.

  password mot de passe transmis en clair.

  crypt mot de passe transmis en chiffré

- Connexions distantes, mode supplémentaire :

  ident authentification TCP/IP de l'utilisateur Unix.

Connexions à un serveur Postgres

- Connexions locales (`locahost`), plusieurs modes :

trust	les connexions sont autorisées sans condition.
reject	les connexions sont refusées sans condition.
password	mot de passe transmis en clair.
crypt	mot de passe transmis en chiffré

- Connexions distantes, mode supplémentaire :

ident	authentification TCP/IP de l'utilisateur Unix.

- Fichiers de configuration (`pg_hba.conf`) pour préciser machine(s) éligibles

La sécurité sous Postgres

- Plusieurs niveaux d'utilisateurs :

La sécurité sous Postgres

- Plusieurs niveaux d'utilisateurs :
  - L'utilisateur **"postgres"** dispose de tous les droits (root), création bases, utilisateurs, destruction,...

La sécurité sous Postgres

- Plusieurs niveaux d'utilisateurs :
  - L'utilisateur **"postgres"** dispose de tous les droits (root), création bases, utilisateurs, destruction,...
  - Les **utilisateurs-administrateurs** : peuvent créer des bases, peuvent autoriser d'autres utilisateurs à accéder à ces bases.
    Certains utilisateurs-administrateurs peuvent créer d'autres utilisateurs.

La sécurité sous Postgres

- Plusieurs niveaux d'utilisateurs :
  - L'utilisateur **"postgres"** dispose de tous les droits (root), création bases, utilisateurs, destruction,...
  - Les **utilisateurs-administrateurs** : peuvent créer des bases, peuvent autoriser d'autres utilisateurs à accéder à ces bases.
    Certains utilisateurs-administrateurs peuvent créer d'autres utilisateurs.
  - Les **utilisateurs** peuvent accéder à des bases (selon les droits)

La sécurité sous Postgres

- Plusieurs niveaux d'utilisateurs :
  - L'utilisateur **"postgres"** dispose de tous les droits (root), création bases, utilisateurs, destruction,...
  - Les **utilisateurs-administrateurs** : peuvent créer des bases, peuvent autoriser d'autres utilisateurs à accéder à ces bases.
    Certains utilisateurs-administrateurs peuvent créer d'autres utilisateurs.
  - Les **utilisateurs** peuvent accéder à des bases (selon les droits)
- Niveau défini lors de la création d'utilisateur (`create user/role`) et évolutif (`alter user/role`)

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)
- `pg_user` est accessible à tous mais pas de mot de passe affiché.

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)
- `pg_user` est accessible à tous mais pas de mot de passe affiché.
- `pg_shadow` est accessible aux administrateurs.

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)
- `pg_user` est accessible à tous mais pas de mot de passe affiché.
- `pg_shadow` est accessible aux administrateurs.
- Impossible de connaître le mot de passe : le champ `pg_shadow.passwd` est crypté.

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)
- `pg_user` est accessible à tous mais pas de mot de passe affiché.
- `pg_shadow` est accessible aux administrateurs.
- Impossible de connaître le mot de passe : le champ `pg_shadow.passwd` est crypté.
- La vérification consiste à crypter un mot de passe saisi puis de le comparer à `pg_shadow.passwd`

Mot de passe des utilisateurs sous Postgres

- Gestion des utilisateurs (vue système `pg_user` de la table `pg_shadow`)
- `pg_user` est accessible à tous mais pas de mot de passe affiché.
- `pg_shadow` est accessible aux administrateurs.
- Impossible de connaître le mot de passe : le champ `pg_shadow.passwd` est crypté.
- La vérification consiste à crypter un mot de passe saisi puis de le comparer à `pg_shadow.passwd`
- L'utilisateur, son administrateur ou 'postgres' peuvent modifier le mot de passe (`alter user/role`).

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :
  - Un groupe d'utilisateurs est un rôle

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :
  - Un groupe d'utilisateurs est un rôle
  - Un utilisateur est un rôle

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :
  - Un groupe d'utilisateurs est un rôle
  - Un utilisateur est un rôle
  - mais un rôle n'est pas forcément un utilisateur (s'il ne dispose pas de droit de login)

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :
  - Un groupe d'utilisateurs est un rôle
  - Un utilisateur est un rôle
  - mais un rôle n'est pas forcément un utilisateur (s'il ne dispose pas de droit de login)
- Un utilisateur peut avoir plusieurs rôles

Les rôles sous Postgres

- On peut regrouper les utilisateurs par **rôle**.
- Vue système `pg_roles`.
- Sous Postgres, "tout est rôle" :
  - Un groupe d'utilisateurs est un rôle
  - Un utilisateur est un rôle
  - mais un rôle n'est pas forcément un utilisateur (s'il ne dispose pas de droit de login)
- Un utilisateur peut avoir plusieurs rôles
- Hiérarchie de rôles

Définition des rôles Postgres

- Un administrateur peut créer/supprimer des rôles (create/drop).

Définition des rôles Postgres

- Un administrateur peut créer/supprimer des rôles (create/drop).
- Syntaxe : `create role user [ [with] option { option }]`

Définition des rôles Postgres

- Un administrateur peut créer/supprimer des rôles (create/drop).
- Syntaxe : `create role user [ [with] option { option }]`
- Les options possibles sont :

Nom	Signification
`LOGIN`	Permet au rôle de se connecter à une base
`SUPERUSER`	Permet au rôle d'être super utilisateur
`CREATEDB`	Permet au rôle de créer des bases
`CREATEROLE`	Permet au rôle de créer des rôles
`PASSWORD`	assigne un mot de passe, exemple : `create role dupont password 'secret'`
`INHERIT` ou `NOINHERIT`	Permet ou pas d'hériter des droits des autres rôles

Définition des rôles Postgres

- Un administrateur peut créer/supprimer des rôles (create/drop).
- Syntaxe : `create role user [ [with] option { option }]`
- Les options possibles sont :

Nom	Signification
`LOGIN`	Permet au rôle de se connecter à une base
`SUPERUSER`	Permet au rôle d'être super utilisateur
`CREATEDB`	Permet au rôle de créer des bases
`CREATEROLE`	Permet au rôle de créer des rôles
`PASSWORD`	assigne un mot de passe, exemple : `create role dupont password 'secret'`
`INHERIT` ou `NOINHERIT`	Permet ou pas d'hériter des droits des autres rôles

- La commande `alter role` permet de modifier ces options.

Affectation des rôles Postgres

- Assigner les droits du rôle `r1` au rôle `r2` :
  Exemple :`grant gba to john ;`
  "john" (r2) devient membre du rôle "gba" (r1) et obtient tous les droits attribués à "gba"

Affectation des rôles Postgres

- Assigner les droits du rôle `r1` au rôle `r2` :
  Exemple :`grant gba to john ;`
  "john" (r2) devient membre du rôle "gba" (r1) et obtient tous les droits attribués à "gba"
- Opération duale : retrait des droits :
  Exemple :`revoke gba from john ;`

Affectation des rôles Postgres

- Assigner les droits du rôle `r1` au rôle `r2` :
  Exemple :`grant gba to john ;`
  "john" (r2) devient membre du rôle "gba" (r1) et obtient tous les droits attribués à "gba"
- Opération duale : retrait des droits :
  Exemple :`revoke gba from john ;`
- Possibilité de hiérarchies de rôles :
  Rappel : tout est rôle (membre = user = rôle)
  Exemple :`grant gba to gba4 ;`

Affectation des rôles Postgres

- Assigner les droits du rôle `r1` au rôle `r2` :
  Exemple :`grant gba to john ;`
  "john" (r2) devient membre du rôle "gba" (r1) et obtient tous les droits attribués à "gba"
- Opération duale : retrait des droits :
  Exemple :`revoke gba from john ;`
- Possibilité de hiérarchies de rôles :
  Rappel : tout est rôle (membre = user = rôle)
  Exemple :`grant gba to gba4 ;`
- Rôle particulier : `public` (tout le monde)

Affectation des droits d'une table à un rôle Postgres

- Syntaxe :
  `grant ⟨liste_droits⟩ on ⟨objet⟩ to ⟨role⟩`

Affectation des droits d'une table à un rôle Postgres

- Syntaxe :

  `grant ⟨liste_droits⟩ on ⟨objet⟩ to ⟨role⟩`

- Exemples :

```
grant select on etudiant to gba;
grant select,update,delete on etudiant to secretariat_gba;
grant select,update(note) on etudiant to profs_gba;
```

Affectation des droits d'une table à un rôle Postgres

- Syntaxe :

  `grant ⟨liste_droits⟩ on ⟨objet⟩ to ⟨role⟩`

- Exemples :

```
grant select on etudiant to gba;
grant select,update,delete on etudiant to secretariat_gba;
grant select,update(note) on etudiant to profs_gba;
```

- ⟨objet⟩ désigne une table

Affectation des droits d'une table à un rôle Postgres

- Syntaxe :
  `grant ⟨liste_droits⟩ on ⟨objet⟩ to ⟨role⟩`
- Exemples :

```
grant select on etudiant to gba;
grant select,update,delete on etudiant to secretariat_gba;
grant select,update(note) on etudiant to profs_gba;
```

- ⟨objet⟩ désigne une table
- Possibilité de préciser une colonne ($3^{ième}$ exemple)

Quelques droits usuels de table (1/2)

SELECT Autorise `SELECT` sur toutes les colonnes, ou sur les colonnes listées spécifiquement, de la table, vue ou séquence indiquée.

Quelques droits usuels de table (1/2)

SELECT Autorise `SELECT` sur toutes les colonnes, ou sur les colonnes listées spécifiquement, de la table, vue ou séquence indiquée.

INSERT Autorise `INSERT` d'une nouvelle ligne dans la table indiquée. Si des colonnes spécifiques sont listées, seules ces colonnes peuvent être affectées dans une commande INSERT, (les autres colonnes recevront par conséquent des valeurs par défaut)

Quelques droits usuels de table (2/2)

UPDATE Autorise `UPDATE` sur toute colonne de la table spécifiée, ou sur les colonnes spécifiquement listées. (En fait, toute commande `UPDATE` non triviale nécessite aussi le droit `SELECT` car elle doit référencer les colonnes pour déterminer les lignes à mettre à jour et/ou calculer les nouvelles valeurs des colonnes.)

Quelques droits usuels de table (2/2)

UPDATE Autorise `UPDATE` sur toute colonne de la table spécifiée, ou sur les colonnes spécifiquement listées. (En fait, toute commande `UPDATE` non triviale nécessite aussi le droit `SELECT` car elle doit référencer les colonnes pour déterminer les lignes à mettre à jour et/ou calculer les nouvelles valeurs des colonnes.)

DELETE Autorise `DELETE` d'une ligne sur la table indiquée. (En fait, toute commande `DELETE` non triviale nécessite aussi le droit `SELECT` car elle doit référencer les colonnes pour déterminer les lignes à supprimer.)

Affectation des droits

- Il existe bien d'autres attributions de droits sur les éléments suivants :

Affectation des droits

- Il existe bien d'autres attributions de droits sur les éléments suivants :
  - vue, séquence, base, fonctions, langages de procédure, schéma.

Affectation des droits

- Il existe bien d'autres attributions de droits sur les éléments suivants :
  - vue, séquence, base, fonctions, langages de procédure, schéma.
- La commande `grant` dispose de l'option `with grant option` : celui qui reçoit le droit peut le transmettre à son tour

Affectation des droits

- Il existe bien d'autres attributions de droits sur les éléments suivants :
  - vue, séquence, base, fonctions, langages de procédure, schéma.
- La commande `grant` dispose de l'option `with grant option` : celui qui reçoit le droit peut le transmettre à son tour
- Quelques contrôles effectués par le SGBD : cycle d'héritage des rôles.

Tables et vues, pas les mêmes droits (1/3)

- Utilisateur "carono", propriétaire de la base :

```
select current_user ;
current_user
--------------
carono
(1 row)

select * from etudiant ;
 login  | adresse
--------+---------
 carono | lille
 demo   | paris
(2 rows)
```

Tables et vues, pas les mêmes droits (2/3)

- Utilisateur "carono", propriétaire de la base :

```
create view my_etudiant as
  select * from etudiant where login=current_user ;
CREATE VIEW

select * from my_etudiant ;
 login  | adresse
--------+---------
 carono | lille
(1 row)

grant select on my_etudiant to demo ;
```

Tables et vues, pas les mêmes droits (3/3)

- Utilisateur "demo" :

```
select * from my_etudiant ;
 login | adresse
-------+---------
 demo  | paris
(1 row)

select * from etudiant ;
ERROR:  permission denied for table etudiant
```

Exercice

Exercice 1

Quelles sont les commandes pour créer un utilisateur `u1` et deux rôles `r1` et `r2`, attribuer le droit `select` d'une table `t` au rôle `r1`, faire hériter les droits de `r1` à `r2`, faire hériter les droits de `r2` à `u1`.

Exercice

Exercice 1

Quelles sont les commandes pour créer un utilisateur `u1` et deux rôles `r1` et `r2`, attribuer le droit `select` d'une table `t` au rôle `r1`, faire hériter les droits de `r1` à `r2`, faire hériter les droits de `r2` à `u1`.

Exercice 2

L'utilisateur `u1` essaye de faire un `select` sur la table `t`. Que se passe-t-il ?

Les types de contraintes

- Normalisation SQL-92

Les types de contraintes

- Normalisation SQL-92
- Les contraintes de domaine
  définissent les valeurs prises par un attribut.

Les types de contraintes

- Normalisation SQL-92
- Les contraintes de domaine
  définissent les valeurs prises par un attribut.
- Les contraintes d'intégrité d'entité
  précisent la clé primaire de chaque table

Les types de contraintes

- Normalisation SQL-92
- Les contraintes de domaine
  définissent les valeurs prises par un attribut.
- Les contraintes d'intégrité d'entité
  précisent la clé primaire de chaque table
- Les contraintes d'intégrité référentielle
  assurent la cohérence entre les clés primaires et les clés étrangères

Les types de contraintes

- Normalisation SQL-92
- Les contraintes de domaine
  définissent les valeurs prises par un attribut.
- Les contraintes d'intégrité d'entité
  précisent la clé primaire de chaque table
- Les contraintes d'intégrité référentielle
  assurent la cohérence entre les clés primaires et les clés étrangères
- Les assertions
  spécifient des contraintes plus générales entre attributs quelconques.

Contrainte de domaine : NOT NULL

```
CREATE TABLE personnel (
    nom TEXT NOT NULL,
    prenom TEXT
) ;

INSERT INTO personnel(nom) VALUES ('dupont')

INSERT INTO personnel(prenom) VALUES('henri')   -> ERREUR
```

Contrainte de domaine : DEFAULT

```
CREATE TABLE article (
   num  INT  NOT NULL,
   quantite INT DEFAULT 1,
   date_creation DATE DEFAULT now()
) ;
```

Contrainte de domaine : UNIQUE

- Éviter les redondances :

```
CREATE TABLE article (
  num  INT  NOT NULL UNIQUE,
  nom TEXT  ...
```

- L'unicité peut être constituée de plusieurs attributs :

```
CREATE TABLE reserve_par (
  num_client INT NOT NULL,
  num_livre  INT NOT NULL,
 UNIQUE (num_client, num_livre)
) ;
```

Contrainte de domaine : CHECK (1/2)

- But : spécifier une contrainte qui doit être vérifiée à tout moment par les tuples de la table :

```
CREATE TABLE personnel (
  num  INT  NOT NULL UNIQUE,
  age INT CHECK (age >= 18),
  sexe CHAR DEFAULT 'F' CHECK (sexe IN ('M','F')),
  ageFuturePromotion INT CHECK (ageFuturePromotion >= age)
) ;
```

Contrainte de domaine : CHECK (2/2)

- La clause CHECK peut se placer après la définition de tous les attributs.
  - Il est préférable de nommer la contrainte (facultatif)
- Utilisation de sous-requêtes SQL

```
CONSTRAINT moy_age
 CHECK ((select avg(age) from personnel) > 35))
```

Déclaration d'un domaine

- Plusieurs attributs ont le même type et les mêmes contraintes
- Syntaxe :

```
CREATE DOMAIN nom [AS] type_donnees
  [DEFAULT expression ] [ contrainte [ ...] ]
```

- Exemple :

```
CREATE DOMAIN entier_positif
   INT DEFAULT 0 CHECK (VALUE >=0) ;

CREATE TABLE personnel(
   num  INT  NOT NULL UNIQUE,
   age entier_positif
   ...
```

Les contraintes d'intégrité d'entité

- Permet de spécifier la clé primaire
- Analogue à NOT NULL UNIQUE
- Génère un index
- Peut être spécifiée à part lorsque la clé est constituée de plusieurs attributs (idem clause UNIQUE)

```
CREATE TABLE personnel (num INT PRIMARY KEY
    ...
```

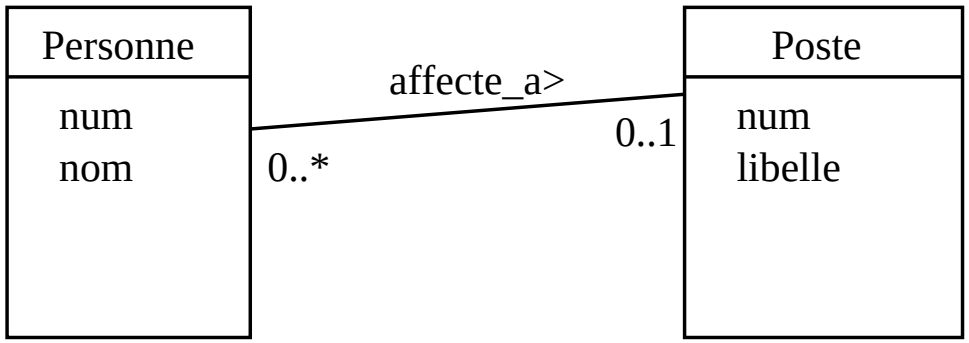
Les contraintes d'intégrité référentielle

- Définies dans la norme SQL-92

Les contraintes d'intégrité référentielle

- Définies dans la norme SQL-92
- Permettent d'assurer la cohérence des associations issues de la conception.

CIR : une cardinalité "0..1" (1/3)



CIR : une cardinalité "0..1" (2/3)

- Réalisation des tables :

```
CREATE TABLE poste (
    num INT PRIMARY KEY,
    libelle TEXT NOT NULL UNIQUE
) ;
CREATE TABLE personne (
    num INT PRIMARY KEY,
    nom TEXT NOT NULL,
    num_poste INT REFERENCES poste) ;
```

CIR : une cardinalité "0..1" (3/3)

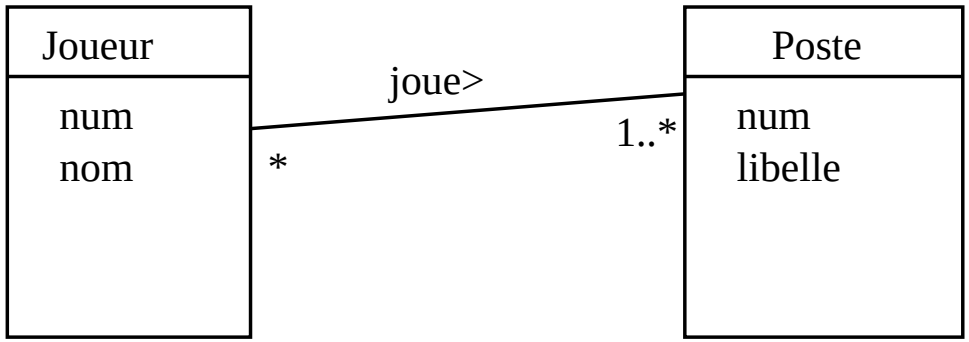
- table poste :

num	libelle
1	'directeur'
2	'ingenieur'
3	'agent'

- exécution :

```
INSERT INTO personne VALUES (1,'dupont', 1);      -> OK
INSERT INTO personne VALUES (2,'durant', 4);      -> ERREUR
DELETE FROM poste  WHERE num=1;                   -> ERREUR
UPDATE personne SET num_poste=NULL WHERE num=1;   -> OK
DELETE FROM poste WHERE num=1;                    -> OK
```

CIR : une cardinalité "*" (1/3)



CIR : une cardinalité "*" (2/3)

- Réalisation des tables :

```
CREATE TABLE poste (
   num INT PRIMARY KEY,
   libelle TEXT NOT NULL UNIQUE
) ;
CREATE TABLE joueur (
 num INT PRIMARY KEY, nom TEXT NOT NULL
);
CREATE TABLE joue (
   num_joueur INT NOT NULL REFERENCES joueur,
   num_poste INT NOT NULL REFERENCES poste,
   PRIMARY KEY (num_joueur, num_poste)
);
```

CIR : une cardinalité "*" (3/3)

- Les tables joueur et poste :

joueur	
num	nom
1	'Tchouameni'
2	'Maignan'

poste	
num	libelle
1	'goal'
2	'defenseur'
3	'milieu'
4	'attaquant'

- Exécution (en rouge : erreur) :

```
INSERT INTO joue(num_joueur,num_poste) VALUES (1,2);
INSERT INTO joue(num_joueur,num_poste) VALUES (1,3);
INSERT INTO joue(num_joueur,num_poste) VALUES (2,1);
```

```
INSERT INTO joue(num_joueur,num_poste) VALUES (2,1)
INSERT INTO joue(num_joueur,num_poste) VALUES (2,5)
DELETE FROM poste where num=3
DELETE FROM poste where num=4
```

Contraintes et clés étrangères

- Plusieurs modes possibles.

Contraintes et clés étrangères

- Plusieurs modes possibles.
- Objectifs communs : préserver la cohérence de la base

Contraintes et clés étrangères

- Plusieurs modes possibles.
- Objectifs communs : préserver la cohérence de la base
- Mode par défaut :
  Refuser l'opération si contrainte non respectée

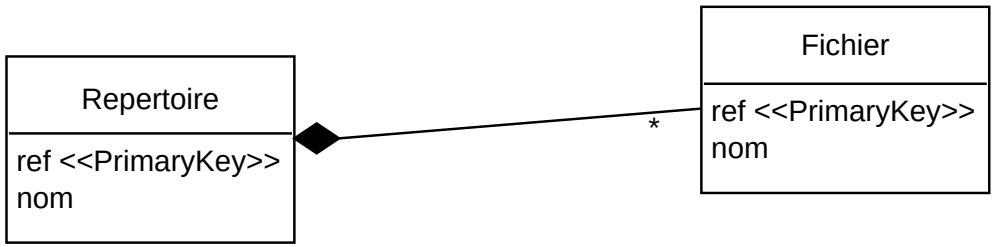
Contraintes et clés étrangères

- Plusieurs modes possibles.
- Objectifs communs : préserver la cohérence de la base
- Mode par défaut :
  Refuser l'opération si contrainte non respectée
- Autre mode :
  Accepter l'opération et modification en cascade pour préserver la cohérence.

Contraintes et clés étrangères

- Plusieurs modes possibles.
- Objectifs communs : préserver la cohérence de la base
- Mode par défaut :
  Refuser l'opération si contrainte non respectée
- Autre mode :
  Accepter l'opération et modification en cascade pour préserver la cohérence.
- Définition des modes lors de la définition de contraintes.

Contraintes sur clés étrangères : un exemple



- Valeurs initiales des tables :

Table repertoire		Table fichier		
ref	nom	ref	nom	ref_rep
0	'/'	1	'fic1'	1
1	'/rep1'	2	'fic2'	2
2	'/rep2'	3	'fic3'	1

Contraintes sur clé étrangère (1/5)

- Mode par défaut
- Le plus restrictif : refus de l'opération
- Exemple :

```
CREATE TABLE fichier (
    ref INTEGER PRIMARY KEY, nom TEXT NOT NULL,
    ref_rep INTEGER REFERENCES repertoire
            ON DELETE RESTRICT
) ;
...
DELETE FROM repertoire where ref=1 ;
```

- Résultat : la suppression est refusée

Contraintes sur clé étrangère (2/5)

- Le mode permissif
- Exemple :

```
CREATE TABLE fichier (
    ref INTEGER PRIMARY KEY, nom TEXT NOT NULL,
    ref_rep INTEGER REFERENCES repertoire
            ON DELETE CASCADE
) ;
...
DELETE FROM repertoire WHERE ref=1 ;
```

- Résultat : '/rep1' supprimé, 'fic1' et 'fic3' également !
- Cette version est la plus conforme au schéma UML (lien de composition)

Contraintes sur clé étrangère (3/5)

- Valeurs par défaut
- Exemple :

```
CREATE TABLE fichier (
    ref INTEGER PRIMARY KEY, nom TEXT NOT NULL,
    ref_rep INTEGER DEFAULT 0 REFERENCES repertoire
            ON DELETE SET DEFAULT
) ;
DELETE FROM repertoire WHERE ref=1 ;
```

- Résultat : supprime '/rep1', 'fic1' et 'fic3' sont désormais dans '/'
- Si la valeur par défaut est incohérente, la suppression est refusée

Contraintes sur clé étrangère (4/5)

- Clause `SET NULL`
- Exemple :

```
CREATE TABLE fichier (
    ref INTEGER PRIMARY KEY, nom TEXT NOT NULL,
    ref_rep INTEGER REFERENCES repertoire
            ON DELETE SET NULL
) ;
...
DELETE FROM repertoire WHERE ref=1 ;
```

- Résultat : supprime '/rep1', 'fic1' et 'fic3' ne sont pas affecté à un répertoire

Contraintes sur clé étrangère (5/5)

- Les contraintes sont également valables pour la mise à jour
- Syntaxe identique
- Exemple :

```
CREATE TABLE fichier (
    ref INTEGER PRIMARY KEY, nom TEXT NOT NULL,
    ref_rep INTEGER REFERENCES repertoire
            ON DELETE CASCADE ON UPDATE CASCADE
) ;
...
```

Conclusion contraintes

- Mécanisme de contraintes très développé
- Largement utilisé depuis SQL-92
- Syntaxe classique (contraintes nommées) :

  CONSTRAINT nom [**UNIQUE** | **NOT NULL** | ...]

- Mise à jour de contraintes via `ALTER TABLE`

Pour gérer les accès concurrents : définir des transactions

- Le concept de transaction va permettre de définir des processus garantissant que l'état de la base est toujours **cohérent**
  - Même en cas d'accès concurrents à la base.
  - Même en cas de panne logicielle ou matérielle.
- Plan du cours :
  - Présentation des problèmes rencontrés (si pas de gestion de transactions)
  - Concept de transaction

Exemple

- On considère le célèbre exemple de virement bancaire :

```
procedure virement(A,B,X) {
 A := A-X ;
 B := B+X ;
}
```

- A et B sont des "entités" de la base de donnée, X est la valeur du virement.
- A := A-X est une façon simplifiée d'écrire :
  `update COMPTE set solde = solde-X`
  `where refCompte = refA ;`
- Par la suite, un appel à cette procédure se fera à l'intérieur d'une *transaction*.

Lecture/Ecriture

Pour mettre en évidence les problèmes, on s'intéressera aux lectures et écritures faites par une séquence d'instructions. La procédure de virement devient alors :

```
debut virement
lire(A)
ecrire(A)
lire(B)
ecrire(B)
fin virement
```

Premier problème

Un utilisateur exécute un virement bancaire :

```
debut virement
lire(A)
ecrire(A)
PANNE SYSTEME
```

- La base se retrouve dans un état incohérent (on a écrit A et pas B)
- Besoin d'un système transactionnel :
  - doit garantir que la transaction se fait complètement ou pas du tout,
  - doit donc annuler la modification de A.
  - Une séquence d'instructions dans une transaction est *Atomique*.

Accès concurrents

Deux utilisateurs exécutent des virements de mêmes comptes, à l'aide de deux séquences d'instructions S1 et S2 :

```
S1 : virement(A,B,100)
S2 : virement(A,B,200)
```

Pour des raisons de performance, les actions des différentes séquences sont entrelacées.
Il faut différencier les actions de S1 de celles de S2, et considérer l'ordre dans lequel ces actions vont s'exécuter.
Pour cet exemple, les soldes de départ de A et B sont respectivement de 500 et 300 Euros.

Ordonnancement

Un ordonnancement est une séquence d'actions de la forme (nomSéquence, opération, donnée).

Table – Exemple d'ordonnancement O_1 de S1 et S2

Ordonnancement	**Action**	**Solde A**	**solde B**
(S1, lire, A)	lire A :500	500	300
(S1, ecrire, A)	écrire A :500-100	400	300
(S2, lire, A)	lire A :400	400	300
(S2, ecrire, A)	écrire A :400-200	200	300
(S1, lire, B)	lire B :300	200	300
(S1, ecrire, B)	écrire B :300+100	200	400
(S2, lire, B)	lire B :400	200	400
(S2, ecrire, B)	ecrire B :400+200	200	600

Deuxième problème

Table – Exemple d'ordonnancement O_2 de S1 et S2

Ordonnancement	Action	Solde A	solde B
(S1, lire, A)	lire A :500	500	300
(S2, lire, A)	lire A :500	500	300
(S1, ecrire, A)	écrire A :500-100	400	300
(S1, lire, B)	lire B :300	400	300
(S1, ecrire, B)	écrire B :300+100	400	400
(S2, ecrire, A)	écrire A :500-200	300	400
(S2, lire, B)	lire B :400	300	400
(S2, ecrire, B)	ecrire B :400+200	**300**	600

Ici, la base est dans un état *inconsistant*. Les effets des séquences sont modifiés à cause de la *concurrence*.

Autre problème : lectures non consistantes

Table – exemple lectures successives

Séquence 1		Séquence 2	
Action	**Résultat**	**Action**	**Résultat**
lire(A)	affiche A :500		
		écrire(A := 200)	A vaut 200
lire(A)	affiche A :200 !		

Ici, on lit 2 fois une même donnée A (séquence 1) et on obtient des résultats différents.

Propriétés des ordonnancements (1/2)

- Deux actions d'un ordonnancement sont *conflictuelles* si elles concernent la même entité et qu'au moins l'une des deux est une écriture.
- Deux ordonnancements O_1 et O_2 des mêmes séquences sont *équivalents* si pour toutes actions conflictuelles a et a', a est avant a' dans O_1 ssi a est avant a' dans O_2.

Propriétés des ordonnancements (2/2)

- Un ordonnancement est *sérialisable* s'il existe un ordre entre les séquences tel que toutes les opérations d'une séquence S qui sont en conflit avec les opérations d'une autre séquence S', sont exécutées après ces opérations.
- Seuls les ordonnancements *sérialisables* sont corrects.

Exemple

- L'ordonnancement O_1 est sérialisable car équivalent à `S1;S2`.
- L'ordonnancement O_2 n'est pas sérialisable.

Synthèse : concept de transaction (1/2)

Pas de gestion de transactions $\longrightarrow$ problèmes

Les exemples précédents ont exposé les problèmes si les SGBD ne **gèrent pas** les transactions.

- Une transaction est une séquence d'instructions qui modifie la base de données et forme une unité de traitement.
- Elle doit (devrait) respecter les propriétés **ACID**

Atomicité	Une transaction s'effectue entièrement ou pas du tout
Consistance	Une transaction qui prend la base dans un état cohérent doit la rendre dans un état cohérent.
Isolement	Pas d'interférence avec les utilisateurs concurrents.
Durabilité	Les actions effectuées par une transaction terminée sont prises en compte dans la base de données.

Synthèse : concept de transaction (2/2)

- Les transactions sont gérées par un moniteur transactionnel.
- Une transaction peut être dans différents états :
  - Active : pendant le déroulement du programme, tant qu'aucun problème n'apparaît.
  - Partiellement validée : Lorsque la dernière instruction a été atteinte
  - Validée : Après une exécution totalement terminée
  - Echouée : après un problème qui a interrompu la transaction

Instructions SQL de base

- Débuter une transaction : `begin`, mémorisation de l'état supposé cohérent de la base.
- Valider une transaction : `commit`.
  Cette instruction permet de signaler que la transaction s'est bien terminée, les modifications qu'elle a effectuées sont rendues visibles aux autres transactions.
- Annuler *explicitement* une transaction : `rollback` ou `abort`. Toutes les commandes depuis le début de la transaction sont annulées.
- Si une transaction échoue (erreur syntaxe SQL, droits, ...), un `rollback` *implicite* permet d'annuler toutes les actions effectuées par cette transaction. La transaction est automatiquement arrêtée.

Isolation des transactions (1/2)

- Le standard SQL définit 4 niveaux d'isolation des transactions.
- Les niveaux diffèrent par les **phénomènes** :

Lecture sale Une transaction lit les données écrites par une transaction concurrente non validée (dirty read)

Lecture non reproductible Une transaction relie des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale) (non repeatable read).

Lecture fantôme Une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée (phantom read).

Isolation des transactions (2/2)

- Plus le niveau est strict, moins les phénomènes sont possibles.
- Le niveau le plus strict est `Serializable` qui doit garantir que l'exécution concurrente des transactions doit produire le même effet que l'exécution consécutive de chacune d'entre elles.

Niveau d'isolation	**Lecture sale**	**lecture non reproductible**	**lecture fantôme**
Uncommited Read : lecture de données non validées	possible pas sous Postgres	possible	possible
Commited Read : lecture de données validées	Impossible	possible	possible
Repeatable Read : lecture répétée	Impossible	Impossible	possible pas sous Postgres
Serializable : sérialisable	Impossible	Impossible	Impossible

Niveaux d'isolation des transactions sous Postgres

- La commande `set transaction` permet de fixer le niveau au sein d'une transaction (après instruction `begin`)
  Syntaxe Postgres : `set transaction isolation level` *niveau*
  niveau peut correspondre à : `SERIALIZABLE`, `REPEATABLE READ`, `READ COMMITTED` ou `READ UNCOMMITTED`
- Le niveau `read commited` est le niveau par défaut.
- Le standard SQL accepte qu'un niveau soit plus strict que la norme, c'est le cas pour Postgres pour les niveaux `Uncommited read` et `Repeatable read`. Cela implique que les "dirty read" sont impossibles sous Postgres.

Illustration des phénomènes

Soit la table SQL `Emp` contenant une seule ligne au départ :

Table – table `Emp`

Id	**nom**
1	dupont

Exemple de lecture sale ("dirty read") et lecture non reproductible

- Rappel : ce cas est impossible sous Postgres

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation level read uncommited		set transaction isolation level read uncommited	
select * from emp	(1, 'dupont')	insert into emp values(3,'durant')	insert 1
select * from emp	(1, 'dupont'), (3,'durant')		
		commit	
select * from emp	(1, 'dupont'), (3,'durant')		
commit			

Exemple de lecture non reproductible

- Sous Postgres, la commande `set transaction` de cet exemple est facultatif (`Read commited` est le mode par défaut)

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation level read commited		set transaction isolation level read commited	
select * from emp	(1,'dupont'),(3,'durant')	insert into emp values(4,'dubois')	insert 1
select * from emp	(1,'dupont'),(3,'durant')	commit	
select * from emp	(1,'dupont'),(3,'durant'), (4,'dubois')		
commit			

Exemple de lecture fantôme

- Rappel : ce cas est impossible sous Postgres avec `repeatable read`

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
set transaction isolation level repeatable read		set transaction isolation level repeatable read	
select * from emp where id between 1 and 3	(1,'dupont'),(3,'durant')	insert into emp values(2,'garcia')	insert 1
select * from emp where id between 1 and 3	(1,'dupont'),(3,'durant')	commit	
select * from emp between 1 and 3	(1,'dupont'),(2,'garcia'), (3,'durant')		
commit			

Exemple Utilisation classique

- Mise en œuvre des transactions par mécanismes de verrouillage de tables ou de lignes selon les cas.

Transaction 1		Transaction 2	
Action	**Résultat**	**Action**	**Résultat**
begin		begin	
select * from emp	(1,'dupont'),(2,'garcia')	select * from emp	(1,'dupont'),(2,'garcia')
		insert into emp	
		values(3,'smith')	insert 1
update temp set			
nom='dupond' where id =1	Update 1		
select * from emp	(1,'dupond', (2,'garcia')	update temp set	
		nom = 'dup' where id=1	bloqué !
commit			update 1
		commit	

- Cette expérience indique l'utilisation de verrous sur les lignes modifiées/créées.

Conclusion transactions

- Bien d'autres possibilités (ex : verrouiller toute une table (`lock table`)
- La gestion des transactions est complexe.
- Les solutions d'implémentation diffèrent d'un éditeur de SGBD à un autre.
- Tout n'est pas standardisé (distinctions entre SGBD).
- La version sûre "Serializable" n'est pas la plus performante.
- Besoin de trouver le bon compromis fiabilité/performances
- Nécessite de comprendre finement les rouages du SGBD