BDW - Conception des BD - modélisation physique

Fabien Duchateau

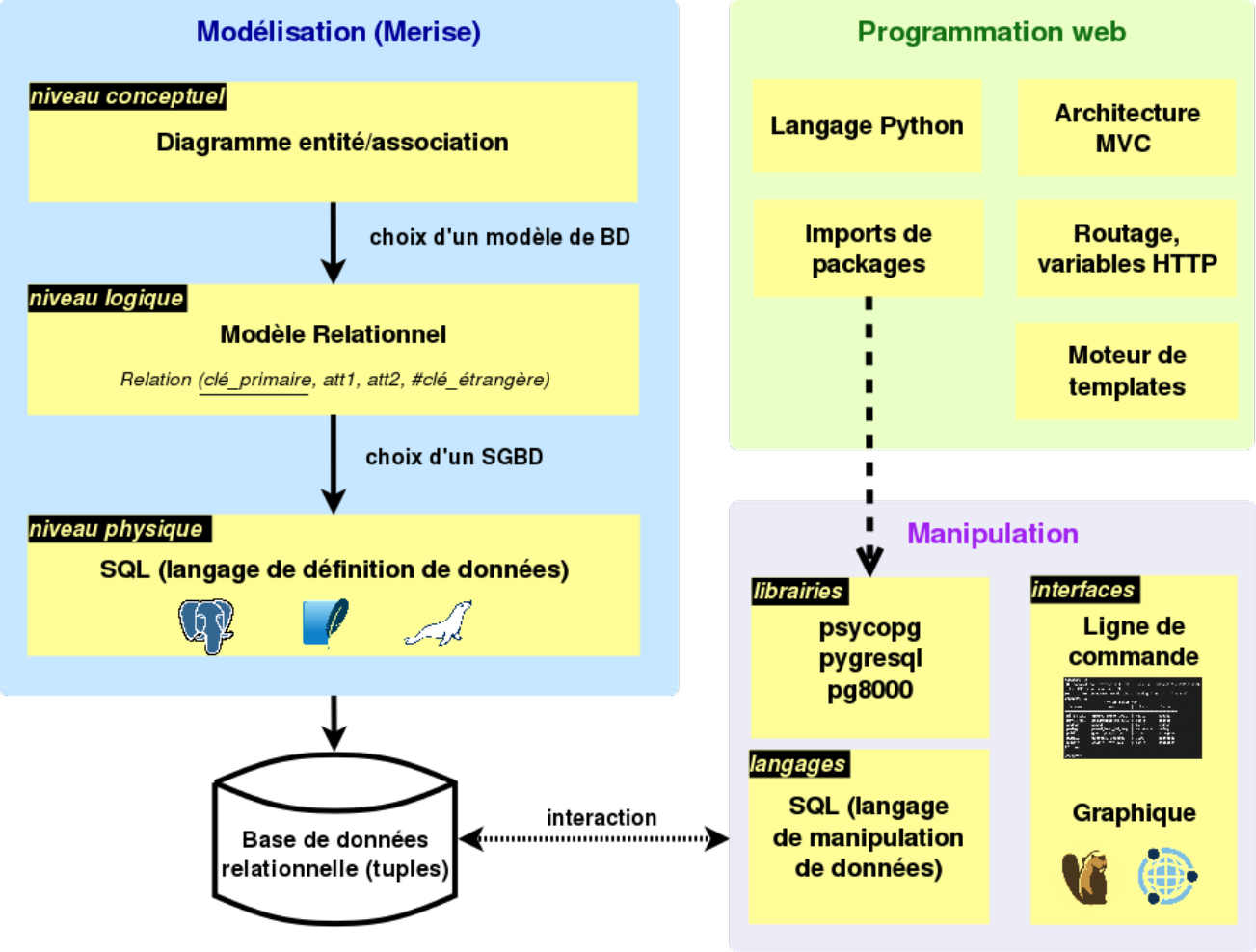
fabien.duchateau [at] univ-lyon1.fr

Université Claude Bernard Lyon 1

2024 - 2025

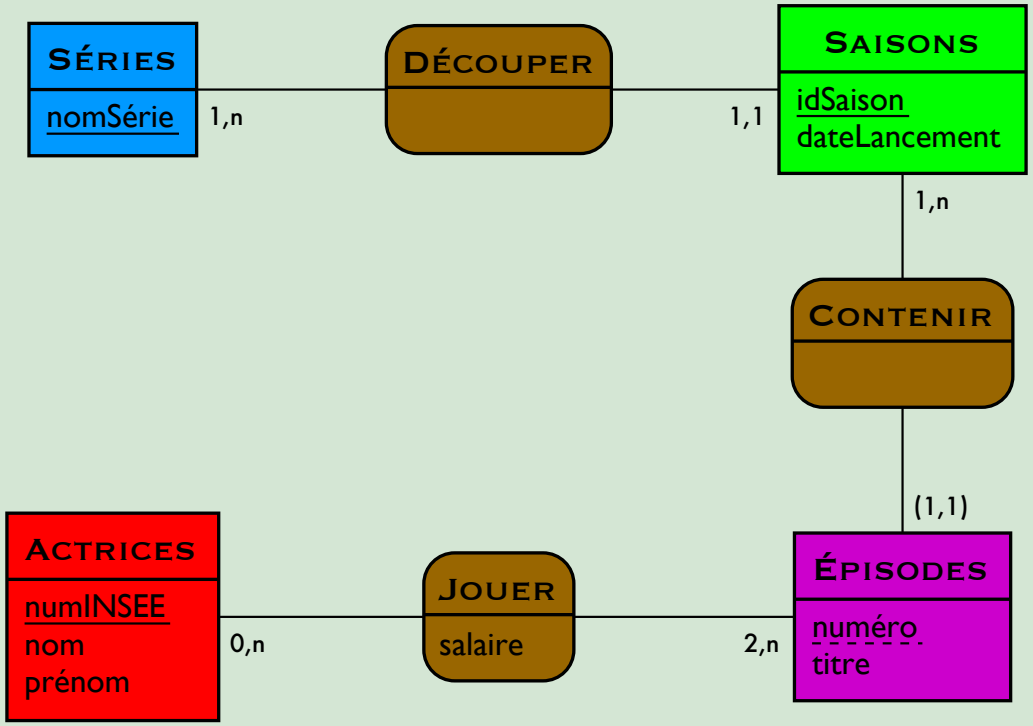
https://perso.liris.cnrs.fr/fabien.duchateau/BDW/

Positionnement dans BDW



Ces diapositives utilisent le genre féminin (e.g., chercheuse, développeuses) plutôt que l'écriture inclusive (moins accessible, moins concise, et pas totalement inclusive)

D'un diagramme E/A à un schéma relationnel



Séries (nomSérie)
Saisons (idSaison, dateLancement, #nomSérie)
Épisodes (numéro, #idSaison, titre)
Actrices (numINSEE, nom, prénom)
Jouer (#numéro, #idSaison, #numINSEE, salaire)

Motivation

Dernière étape de la modélisation (niveau physique) :

- Implique le choix d'un SGBD pour stocker la BD
- Besoin de traduire le modèle logique en modèle physique
- Dans notre cas, transformation du schéma relationnel en tables SQL pour le SGBD PostgreSQL

Le langage SQL permet la création (**C**RUD) à la fois de tables et contraintes (LDD) et d'instances (LMD)

Plan

Le langage SQL

BD et schéma

Création de tables

Insertion de n-uplets

Contraintes d'intégrité

Qu'est ce que SQL ?

SQL = Structured Query Language

Un langage concret pour interagir avec le modèle relationnel :

- Un langage de manipulation de données
- Un langage de description de données
- Un langage pour administrer la base, gérer les contrôles d'accès

Langage de manipulation de données **déclaratif** ⇒ description du résultat escompté

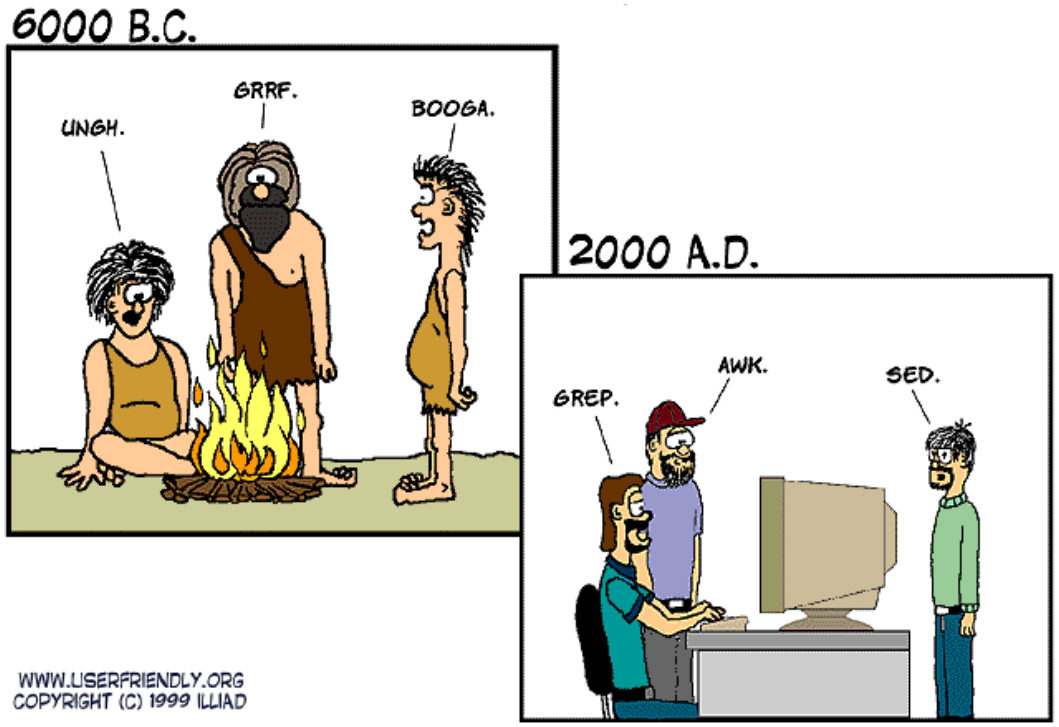
Cours et tutoriels SQL sur http://sql.sh
Voir aussi http://modern-sql.com/ et http://use-the-index-luke.com/

Historique de SQL

Origine : IBM en 1974

Versions de la norme :

- SQL-87 : 1987 (ISO)
- SQL-2 : 1992
- SQL:1999
- ...
- SQL:2019
- SQL:2023



http://fr.wikipedia.org/wiki/SQL

SQL dans les SGBD

De nombreux SGBD utilisent le langage SQL :

- PostgreSQL
- MySQL / MariaDB
- Oracle
- SQL Server
- LibreOffice Base, Access
- ...

Chaque SGBD implémente plus ou moins respectueusement le standard SQL (différents niveaux de conformité)

https://tdan.com/is-sql-a-real-standard-anymore/4923

Plan

Le langage SQL

BD et schéma

Création de tables

Insertion de n-uplets

Contraintes d'intégrité

Création d'une base de données

Dans un SGBD, les tables sont stockées dans une BD :

- Création d'une BD avec une requête SQL ou un client graphique (e.g., DBeaver, PHPMyAdmin)
- Options : jeu de caractères (e.g.,*utf8*), collation (e.g., *utf8_general_ci*)

```
1 CREATE DATABASE p1234567;
```

Dans le cadre de l'UE, vous disposez déjà d'une (seule) base de données nommée p1234567 sur le serveur bd-pedago

https://www.postgresql.org/docs/current/sql-createdatabase.html

Notion de schéma

Dans la norme SQL et dans PostgreSQL, une base de données est découpée en **schémas** :

- Similaire à des "répertoires"
- Permet d'organiser les objets (tables, vues, etc.) dans une même base et de les supprimer facilement
- Par exemple, un schéma *tp1* pour le premier TP, un schéma *projet*, etc.
- Par défaut, un schéma *public*

Pour éviter toute confusion avec le schéma relationnel, on parlera dans la suite de **répertoire schéma**

Création d'un répertoire schéma

Suppression du répertoire schéma *series* (et de tous ses objets), puis création d'un schéma *series* :

```
1 DROP SCHEMA IF EXISTS series CASCADE;
2 CREATE SCHEMA IF NOT EXISTS series;
```

Quel répertoire schéma est utilisé par le SGBD ?

- Déterminé par le `search_path` (liste ordonnée de schémas)
- Quand on utilise un objet (sans préciser son schéma), le SGBD le cherche d'abord dans le premier schéma du `search_path`, puis dans le second, etc.

```
1 SET search_path TO series; -- search_path modifié pour "series"
2 SHOW search_path;  -- affiche series
```

---

https://www.postgresql.org/docs/current/ddl-schemas.html
https://www.postgresql.org/docs/current/sql-createschema.html

Plan

Le langage SQL

BD et schéma

Création de tables

Insertion de n-uplets

Contraintes d'intégrité

Généralités

Pour chaque relation du modèle relationnel, création de la table correspondante :

- Avec une requête SQL ou un client graphique
- Attention aux différences d'implémentation de SQL dans les SGBD (PostgreSQL pour cet enseignement)

À la création d'une table, on indique :

- Le nom des attributs
- Le type de chaque attribut
- De manière optionnelle :
  - certaines contraintes d'intégrité
  - des caractéristiques de stockage

Syntaxe de création de table

CREATE TABLE *nom_table* **(**
att$_1$ *type*$_1$
[, att$_2$ *type*$_2$*, ...]*
);

- Création simple d'une table de nom *nom_table*, avec un attribut att_1 de type $type_1$, un attribut att_2 de type $type_2$, etc.
- Les crochets [...] représentent des éléments optionnels
- Le répertoire schéma dans lequel la table est créée est le schéma courant (le premier du `search_path`) ou peut être spécifiée par la notation `nom_schema.nom_table`

https://www.postgresql.org/docs/current/sql-createtable.html
https://www.postgresql.org/docs/current/datatype.html

Types d'attributs

Type pour les **booléens** :

- `BOOLEAN` (valeurs *true*, *false*, *1*, *0*)

Types pour les **chaînes de caractères** :

- `CHAR(longueur)`
  - taille fixe égale à *longueur* (complétion par des espaces)
- `VARCHAR(longueur)`
  - taille variable, jusqu'à *longueur* caractères
- `TEXT`
  - taille variable (jusqu'à 1 Go)

---

https://www.postgresql.org/docs/current/datatype-boolean.html
https://www.postgresql.org/docs/current/datatype-character.html

Types d'attributs (2)

Types **numériques** :

- `INTEGER`, `SMALLINT`, `BIGINT` (entiers de tailles différentes)
- `REAL`, `DOUBLE`
  - 6 ou 15 chiffres, valeurs approximatives
- `NUMERIC(longueur, échelle)`
  - un nombre codé exactement sur *longueur* chiffres, dont *échelle* chiffres après la virgule
  - valeurs exactes (donc coûts de calcul !)
- `SERIAL`, `SMALLSERIAL` (raccourcis)
  - entier positif auto-incrémenté pour servir d'identifiant

---

https://www.postgresql.org/docs/current/datatype-numeric.html

Types d'attributs (3)

Types pour les **dates** :

- `DATE`, une date au jour près (plusieurs formats, dont *YYYY-MM-JJ*)
- `TIME`, un temps (plusieurs formats, dont *HH:MM:SS.sss*)
- `TIMESTAMP`, une date avec heure (plusieurs formats, dont *YYYY-MM-JJ HH:MM:SS.sss*)

Valeurs spéciales : `NOW` (jour et heure courants), `YESTERDAY`, `TODAY`, `TOMORROW`, etc.

---

https://www.postgresql.org/docs/current/datatype-datetime.html

Types d'attributs (4)

Nombreux autres types :

- Données binaires (BYTEA)
- Géométriques (POINT, LINE, POLYGON, etc.)
- Énumérations (liste de valeurs avec CREATE TYPE)

```
CREATE TYPE categories AS ENUM ('comédie', 'SF', 'drame');
```

- Définis par l'utilisateur (DOMAIN)
- Adresses réseau, tableaux, UUID, XML, JSON, etc.

https://www.postgresql.org/docs/current/datatype.html

Exemple de création de tables

SÉRIES(nomSérie)

```
CREATE TABLE SERIES (
  nomsérie VARCHAR(42)
);
```

```
fduchateau=> \d series
                        Table "series.series"
  Column   |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 nomsérie | character varying(42) |           |          |
```

SAISONS(idSaison, dateLancement, #nomSérie)

```
CREATE TABLE SAISONS (
  idsaison INTEGER,
  datelancement DATE,
  nomsérie VARCHAR(42)
);
```

```
fduchateau=> \d saisons
                        Table "series.saisons"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 idsaison      | integer                |           |          |
 datelancement | date                   |           |          |
 nomsérie      | character varying(42) |           |          |
```

Exemple de création de tables (2)

JOUER(#numero, #idSaison, #numINSEE, salaire)

```
CREATE TABLE JOUER (
  idsaison INTEGER,
  numéro INTEGER,
  numinsee INTEGER,
  salaire REAL
);
```

```
fduchateau=> \d jouer
                Table "series.jouer"
  Column   |  Type   | Collation | Nullable | Default
----------+---------+-----------+----------+---------
 idsaison | integer |           |          |
 numéro   | integer |           |          |
 numinsee | integer |           |          |
 salaire  | real    |           |          |
```

En résumé

- Création de la structure d'une table avec `CREATE TABLE`
- Définition de chaque attribut par son nom et son type
- Visualisation du schéma relationnel (graphiquement)

Mais dans la table créée :

- il n'y a aucune instance (donnée)
- aucune contrainte n'a été spécifiée

Plan

Le langage SQL

BD et schéma

Création de tables

Insertion de n-uplets

Contraintes d'intégrité

Généralités

Méthodes pour ajouter des instances :

- Par un outil graphique (DBeaver, pgAdmin, etc.)
- Avec du code SQL, par insertion d'un ou plusieurs n-uplets
- Avec du code SQL, par copie de données existantes

	nomsérie
1	The Big Bang Theo
2	Game of Thrones
3	Breaking Bad
4	The Wire
5	Black Clover
6	The 100
7	Kaamelott
8	Fallout

```
fduchateau=> insert into series values ('Silo');
INSERT 0 1
fduchateau=> select * from series;
      nomsérie
---------------------
 The Big Bang Theory
 Game of Thrones
 Breaking Bad
 The Wire
 Black Clover
 The 100
 Kaamelott
 Fallout
 Silo
(9 rows)
```

Insertion d'une série dans DBeaver (gauche) et avec psql (droite)

Insertion de n-uplets

INSERT INTO *nom_table*(att_1, ..., att_n)
VALUES(val_1, ..., val_n) **;**

- Insertion d'un n-uplet dans la table *nom_table*
- Chaque attribut de att_1, ..., att_n aura la valeur val_1, ..., val_n correspondante (i.e., val_i pour l'attribut att_i)
- Si un attribut de la table *nom_table* n'apparaît pas dans att_1, ..., att_n, alors la valeur du n-uplet pour cet attribut est NULL
- Version raccourcie sans préciser les attributs : chaque attribut (dans l'ordre de création de la table) doit avoir une valeur

https://www.postgresql.org/docs/current/sql-insert.html

Exemples d'insertion de n-uplets

```sql
INSERT INTO SAISONS(idsaison, datelancement, nomsérie) VALUES(1,
↪  '2011-09-22', 'The Big Bang Theory');
INSERT INTO SAISONS VALUES(2, '2012-09-27', 'The Big Bang Theory');
```

```
fduchateau=> select * from saisons;
 idsaison | datelancement |      nomsérie
----------+---------------+--------------------
        1 | 2011-09-22    | The Big Bang Theory
        2 | 2012-09-27    | The Big Bang Theory
(2 rows)
```

```sql
INSERT INTO SAISONS VALUES(3, '2011-04-17', 'Game of Thrones'),
                           (4, '2014-03-19', 'The 100');
```

```
fduchateau=> select * from saisons;
 idsaison | datelancement |      nomsérie
----------+---------------+--------------------
        1 | 2011-09-22    | The Big Bang Theory
        2 | 2012-09-27    | The Big Bang Theory
        3 | 2011-04-17    | Game of Thrones
        4 | 2014-03-19    | The 100
(4 rows)
```

Copie de données existantes

INSERT INTO *nom_table*$(att_1, \ldots, att_n)$
SELECT $e_1, \ldots, e_n$
FROM ... **;**

- Insertion de n-uplets dans la table *nom_table* à partir de données récupérées par la requête SELECT ... FROM
- La requête ne peut pas contenir de ORDER BY (le SGBD détermine l'ordre de stockage des n-uplets)
- Le nom des attributs dans le résultat de la requête n'est pas important : c'est l'ordre des expressions qui compte

Plan

Le langage SQL

BD et schéma

Création de tables

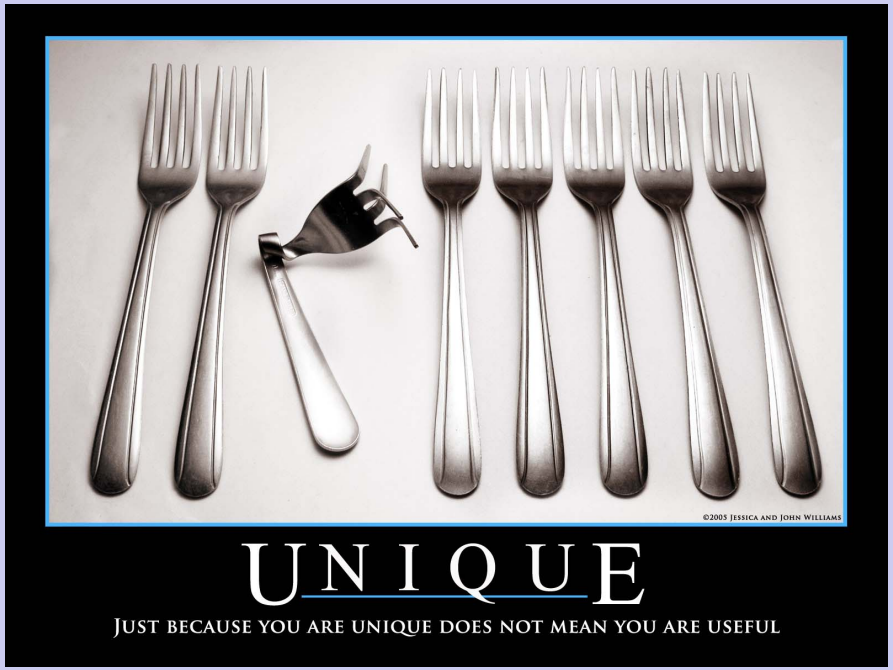
Insertion de n-uplets

Contraintes d'intégrité

Motivation

Dans les exemples précédents, aucune contrainte sur les tables : il est donc possible de créer des épisodes identiques, d'avoir des épisodes sans actrice, d'insérer des salaires négatifs, etc.

Besoin de spécifier des contraintes d'intégrité !



Généralités

Le SGBD vérifie les contraintes lors des insertions, suppressions ou mises à jour de n-uplets

Quand créer les contraintes ?

- Lors de la création de la table (`CREATE TABLE`)
- Lors d'une évolution de la table (`ALTER TABLE`)

Quels types de contraintes ?

- Unicité
- Autorisation des valeurs incconnues (NULL)
- Clé primaire (PK, pour Primary Key)
- Clé étrangère (FK, pour Foreign Key)
- Valeurs autorisées (CHECK)

---

https://www.postgresql.org/docs/current/ddl-constraints.html

Syntaxe de création de contrainte

CREATE TABLE *nom_table* (
att_1 $type_1$,
[...]
CONSTRAINT [nom_1] *def_contrainte*$_1$
[...]
);

- Le mot-clé CONSTRAINT permet de spécifier une contrainte
- nom_1 est le nom de la contrainte (optionnel)
- *def_contrainte*$_1$ définit la contrainte (selon son type)
- Certaines contraintes ont une syntaxe "raccourcie" (contrainte indiquée après l'attribut concerné)

Contrainte UNIQUE

CONSTRAINT nom_c **UNIQUE**(att_i, att_j, ...)

- Avec la contrainte **unique**, chaque n-uplet doit avoir une combinaison de valeurs différente pour les attributs att_i, att_j, ...
- Il est par contre possible d'avoir deux fois la même valeur pour un attribut att_i
- Si l'une des valeurs pour att_i, att_j, ... est NULL, la contrainte ne s'applique pas sur le n-uplet concerné

Contrainte NOT NULL

att_1 $type_1$ [**NOT**] **NULL**

- Contrainte [NOT] NULL uniquement spécifiée lors de la définition d'un attribut
- Indique que l'attribut peut / ne peut pas recevoir de valeurs non renseignées
- Par défaut, les attributs acceptent NULL

Exemples de contrainte NOT NULL et UNIQUE

```sql
CREATE TABLE SERIES (
  nomsérie VARCHAR(42) NOT NULL
); -- impossible d'insérer une série sans spécifier un nom
```

```
fduchateau=> \d series2
                      Table "public.series2"
  Column  |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 nomsérie | character varying(42) |           | not null |
```

```sql
CREATE TABLE SERIES (
  nomsérie VARCHAR(42),
  CONSTRAINT series_unique_nomserie UNIQUE (nomsérie)
);  -- impossible d'avoir 2 séries avec le même nom
```

```
fduchateau=> \d series3
                      Table "public.series3"
  Column  |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 nomsérie | character varying(42) |           |          |
Indexes:
    "series_unique_nomserie" UNIQUE CONSTRAINT, btree ("nomsérie")
```

Contrainte de clé primaire

CONSTRAINT nom_c **PRIMARY KEY**(att_i, att_j, ...)

- Une contrainte de clé primaire indique que l'ensemble d'attributs (att_i, att_j, ...) sert d'identifiant principal pour les n-uplets de la table
- La valeur de la clé primaire permet donc de retrouver un n-uplet de manière non ambigüe
- Implique NOT NULL et UNIQUE sur chacun des (att_i, att_j, ...)
- Au maximum une contrainte PRIMARY KEY par table

Contrainte de clé primaire avec auto-incrément

Il est souvent préférable d'utiliser un entier (sans signification) en clé primaire plutôt qu'un attribut (e.g., le nom de la série, les nom/prénom d'une actrice)

- Un SGBD peut gérer automatiquement ces clés primaires par auto-incrément (type SERIAL pour PostgreSQL)
- Dans la requête d'insertion, on n'utilise pas l'attribut auto-incrémenté
- Le SGBD affecte automatiquement à la clé primaire la plus grande valeur positive de la colonne $+1$
- La première valeur d'un attribut auto-incrémenté est 1

Exemples de contrainte de clé primaire

```sql
CREATE TABLE SERIES (
  nomsérie VARCHAR(42),
  CONSTRAINT series_pk PRIMARY KEY (nomsérie)
);  -- contrainte nommée de clé primaire
```

```
fduchateau=> \d series
                        Table "public.series"
  Column  |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 nomsérie | character varying(42) |           | not null |
Indexes:
    "series_pk" PRIMARY KEY, btree ("nomsérie")
```

```sql
CREATE TABLE SAISONS (
  idsaison SERIAL NOT NULL PRIMARY KEY,  -- contrainte de clé primaire
  ↪  auto-incrémentée
  datelancement DATE,
  nomsérie VARCHAR(42)
);
```

```
fduchateau=> \d saisons
                                          Table "public.saisons"
    Column     |         Type          | Collation | Nullable |                  Default
---------------+-----------------------+-----------+----------+-----------------------------------------
 idsaison      | integer               |           | not null | nextval('saisons_idsaison_seq'::regclass)
 datelancement | date                  |           |          |
 nomsérie      | character varying(42) |           |          |
Indexes:
    "saisons_pkey" PRIMARY KEY, btree (idsaison)
Foreign-key constraints:
    "saisons_nomsérie_fkey" FOREIGN KEY ("nomsérie") REFERENCES series("nomsérie")
```

Contrainte de clé étrangère

CONSTRAINT nom_c **FOREIGN KEY**(att_1, ..., att_k)
REFERENCES $table_cible(att'_1,...,att'_k)$

- Une clé étrangère est une **référence** vers la clé primaire d'une autre table
- Les valeurs pour (att_1, ..., att_k) doivent correspondre aux valeurs d'un des n-uplets de *table_cible* pour ses attributs (att'_1,...,att'_k)

Exemples de contrainte de clé étrangère

```
CREATE TABLE SAISONS (
  idsaison SERIAL NOT NULL PRIMARY KEY,  -- contrainte de clé primaire
  ↪ auto-incrémentée
  datelancement DATE,
  nomsérie VARCHAR(42),
  CONSTRAINT fk_saisons_nomserie FOREIGN KEY (nomsérie) REFERENCES
  ↪ SERIES (nomsérie)  -- contrainte clé étrangère
);
```

```
CREATE TABLE EPISODES (
  idsaison INTEGER REFERENCES SAISONS (idsaison),  -- contrainte de clé
  ↪ étrangère
  numéro INTEGER,
  titre VARCHAR(100),
  PRIMARY KEY (idsaison, numéro)  -- clé primaire composée de 2
  ↪ attributs
);
```

Contrainte CHECK

CONSTRAINT nom_c **CHECK**(*condition*)

- La contrainte CHECK permet de spécifier une condition (booléenne) qui sera vérifiée lors d'insertions ou modification de n-uplets
- La forme CHECK(*att* IN ('val_1', 'val_2', ...)), utilisée pour les types énumérés est un cas particulier de cette contrainte

Exemple de contrainte CHECK

```sql
CREATE TABLE JOUER (
  idsaison INTEGER,
  numéro INTEGER,
  numinsee INTEGER REFERENCES ACTRICES (numinsee),
  salaire REAL CHECK(salaire > 0.0),  -- pas de salaire négatif
  PRIMARY KEY (idsaison, numéro, numinsee),
  CONSTRAINT fk_jouer_saison FOREIGN KEY (idsaison, numéro) REFERENCES
  ↪  EPISODES (idsaison, numéro)   -- clé étrangère avec 2 attributs
);
```

```
fduchateau=> \d jouer
                Table "public.jouer"
  Column  |  Type   | Collation | Nullable | Default
----------+---------+-----------+----------+---------
 idsaison | integer |           | not null |
 numéro   | integer |           | not null |
 numinsee | integer |           | not null |
 salaire  | real    |           |          |
Indexes:
    "jouer_pkey" PRIMARY KEY, btree (idsaison, "numéro", numinsee)
Check constraints:
    "jouer_salaire_check" CHECK (salaire > 0.0::double precision)
Foreign-key constraints:
    "fk_jouer_saison" FOREIGN KEY (idsaison, "numéro") REFERENCES episodes(idsaison, "numéro")
    "jouer_numinsee_fkey" FOREIGN KEY (numinsee) REFERENCES actrices(numinsee)
```

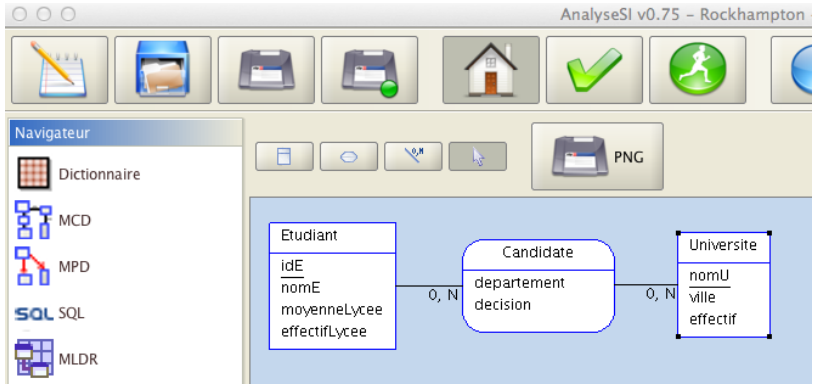
En résumé

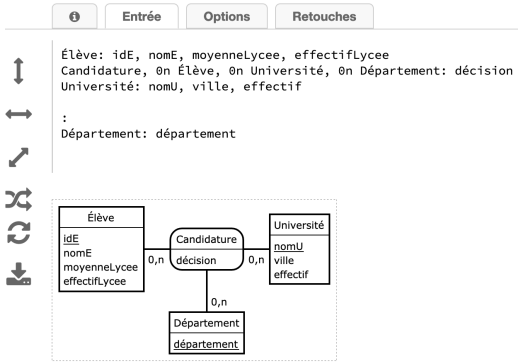
- Contraintes de clés (primaires et étrangères)
- Contraintes sur les valeurs autorisées (UNIQUE, [NOT] NULL, et CHECK)

https://wumo.com/wumo

Rappel sur les outils de modélisation

- **Analyse SI** : https://launchpad.net/analysesi
- **Mocodo** : http://www.mocodo.net/
- **Looping** (Windows) : http://www.looping-mcd.fr/
- JMerise (limité) : http://www.jfreesoft.com/JMerise/
- Open Modelsphere : http://www.modelsphere.org/
- MySQL Workbench : http://mysqlworkbench.org/





Conseils pour la modélisation

Pour modéliser une base de données à partir de spécifications :

- Avec un outil de modélisation :
  - réaliser le diagramme E/A
  - générer le code SQL de création des tables **pour PostgreSQL**
  - vérifier ce qui est généré !
  - créer un autre fichier SQL pour corriger/compléter le code généré
- Peuplement de la BD :
  - créer un autre fichier SQL contenant des requêtes INSERT
  - saisir des instances avec un client graphique, puis exporter dans un fichier

Conseils pour la modélisation (2)

Pourquoi créer plusieurs fichiers ?

- Votre modélisation peut évoluer (erreurs, choix hasardeux) et vous devrez re-générer ou modifier le fichier SQL de création des tables
- Le découpage permet de simplifier la détection (et correction) d'erreurs
- Les instances sont plus faciles à ajouter via un outil graphique (ce qui nécessite un export)