Structure et base de données : synthèse

Langage SQL : Data Definition Language

CREATE DOMAIN

Types fondamentaux de SQL :

- Caractères (et chaînes de caractères) : CHAR(n), VARCHAR(n), LONG VARCHAR, …
- Numériques exacts (signés ou non) : TINYINT, SMALLINT, INT, BIGINT, …
- Numériques décimaux exacts : NUMERIC (p, s) ou DECIMAL (p, s)
- Numériques approximatifs : REAL, DOUBLE, FLOAT
- Logique (boolean) : BIT
- Temporels (dates et heures) : DATE, TIME, TIMESTAMP, …
- Binaires (stockage pur) : BINARY, IMAGE, …
- "Exotiques" : MONEY, SPATIAL, …

CHAR : longueur fixe (32kb max) //char(30)
VARCHAR : longueur variable (32kb max) //varchar(125)

TINYINT : 1 byte non signé (0…255)
[unsigned] SMALLINT : 2 bytes (2^{16} 64k valeurs)
[unsigned] INT : 4 bytes (2^{32} 4G valeurs)
BIGINT : 8 bytes (2^{64} 16E valeurs) //E = Exa = G^2

Les valeurs numériques sont en format « humain », mais les valeurs décimales se marquent avec un point et non une virgule.
Les chaînes de caractères sont **obligatoirement** entre apostrophes simples, il faut le doubler si nécessaire. Exemple : 'hello', 'aujourd''hui'
Les dates sont basiquement sous la forme de chaînes 'aaaa-mm-jj' ou 'aaaa/mm/jj'.
Les heures sont basiquement sous la forme de 'hh:mm:ss:µµµµµµ'.
Il existe également des valeurs spéciales : null, current date, current time, current timestamp, current user, …

Exemple de création de domaine :
CREATE DOMAIN « Dsexe » CHAR(1) NOT NULL DEFAULT 'F' check(@col in ('M','F'));

CREATE TABLE

Exemple de création de table :
CREATE TABLE tbPersonnes(persId int not null autoincrement, persNom varchar(50) not null, persSexe DSexe default 'F', persNaiss date not null);

Une contrainte à **toujours** suivre est d'avoir une clé primaire qui assure l'unicité des occurrences.
Les différentes contraintes d'une table sont : les clés (*primary key*, *unique key*) et les vérifications (*check constraints* entre les colonnes).

Langage SQL : Data Manipulation Language

4 instructions de base :

- SELECT : interrogation de la base de données (lecture)
- INSERT : ajout de nouvelles occurrences dans des tables (écriture)
- UPDATE : modification de valeurs dans des colonnes/lignes (écriture)
- DELETE : suppression d'occurrences (écriture)

SELECT … FROM …

Select select-list from from-expression

select-list : liste (séparé par des virgules) des colonnes de la table dont on veut voir le contenu.

* : signifie toutes les colonnes dans l'ordre existant.

as : système d'alias ou aliasing.

from-expression : spécifie la ou les table(s) à interroger. *Si pas propriétaire de la table, préfixer id-propriétaire.*

- Des expressions peuvent être utilisées avec SELECT :
- Des constantes : nombres, chaînes, dates, heures, …
- Des valeurs spéciales : current date, current user, …
- Des concaténations de chaînes : opérateurs || et +
- Des expressions numériques : +, -, *, /, %
- Des fonctions : des dizaines dans toutes les catégories de types comme left(), year(), …

Toute expression utilisée doit être aliasées (= nommées). Le nommage se fait en camelCase.

Pour ordonner (trier) le résultat, on utilise après select et from : order by order-list
Pour n'obtenir qu'une fois chaque résultat et éviter les doublons, on utilise dans le select : distinct
Pour ajouter des conditions, on utilise après select et from : where search-condition

→ Comparaisons colonne/valeurs (=, !=, <, <=, >, >=) et expressions (NOT, AND, OR).
→ Sucres syntaxiques : between (= entre) et in (= parmi).
→ Autres prédicats : like et « jokers » (% de 0 à n inconnus, _ 1 inconnu à cet endroit).

Fonctions

- ***Concaténation :*** string(str [, …])
  Exemple : select string (vendPrenom, ' ', vendNom) as identite
- ***Suppressions d'espaces :*** ltrim(str), rtrim(str), trim(str)
  l pour left (début) et r pour right (fin)
- ***Extraction de sous-chaîne :***
  left(str, n), right (str, n) : renvoie les n premiers/derniers caractères de str.
  substr(str, start, len) : renvoie len caractères de str à partir de la position start.
  Exemple : select string (left(vendPrenom, 1), '.') as initiale
- ***Changement de casse :*** upper(str), lower(str)
  upper c'est majuscule, lower c'est minuscule
- ***Longueur :*** length(str)
- ***Position de sous-chaîne :*** locate(str1, str2, start)
- ***Formes d'arrondis :*** (numeric)
  ceil(n) : renvoie l'entier supérieur
  floor(n) : renvoie l'entier inférieur
  round(n, dec), truncate (n, dec) : arrondit n vers le haut/bas sur dec décimales.
  truncate coupe à l'unité sans faire attention à ce qu'il y a derrière alors que round fait attention.
- ***Trigonométrie :*** (double)
  sin(n), cos(n), tan(n), cot(n), asin(n), acos(n), atan(n), atan2(n), radian(n), degrees(n)
- ***Algèbre :*** (double)
  exp(n), log(n), log10(n), abs(n), sqrt(n), power(n1, n2)
- ***Conversion en chaîne :*** str(n, len, dec)
  Renvoie le nombre n convertit en chaîne sur len caractères au total et dec chiffres après le point décimal.

Fonctions liées aux dates/temps :

- ***Acquisition :*** date(s), datetime(s), getdate()
- ***Extractions :***
  year(ts), month(ts), day(ts) : renvoie l'année, le mois, le jour de d
  hour(ts), minute(ts), second(ts) : renvoie l'heure, les minutes, les secondes de h.
- ***Spécification de parties :*** datepart(part, ts)
  Exemple : datepart(day, '2015-02-23 15:30') → 23
- ***Opérations :*** timestamp, entier
  dateadd(part, i, ts) : ajoute/soustrait i parties part au timestamp ts
  datediff(part, ts1, ts2) : renvoie la différence entre ts1 et ts2 exprimée en part
  years(ts1, ts2), months(ts1, ts2), days(ts1, ts2), hours(ts1, ts2) : différence entre 2
- ***Formatages :*** string
  dateformat(ts, s) : formate ts en fonction du string s
  monthname(ts), dayname(ts) : renvoie le nom du mois/jour de ts.

Fonctions liées aux nombres :

- ***Formatage de nombres :*** str(n, len, dec)
- ***Conversions :*** cast(expr as datatype), convert(datatype, expr, style)
- ***Traitement du null :***
  isnull(arg-list) : le premier argument non null
  ifnull(arg1, arg2, arg3) : si arg1 is null alors arg2 sinon arg3
  nullif(arg1, arg2) : si arg1 = arg2 alors null sinon arg1

Jointures

4 moyens différents pour la même chose, le même résultat.
2 est la syntaxe light avec alias de tables, 3 la syntaxe light avec jointure naturelle (colonnes de même nom) et 4 est la syntaxe light avec jointure automatique sur clés étrangères/primaire.

```
1 : select *
      from dba.tbVendeurs join dba.tbVilles
          on dba.tbVendeurs.vilId = dba.tbVilles.vilId

2 : select *
      from dba.tbVendeurs as t1 join dba.tbVilles as t2
          on t1.vilId = t2.vilId

3 : select*
      from dba.tbVendeurs natural join dba.tbVilles
ou select*
      from dba.tbVendeurs join dba.tbVilles

4 : select *
      from dba.tbVendeurs key join dba.tbVilles
```

Fonctions agrégatives

Déterminer une valeur (unique) **sur un ensemble de lignes**.

- Comptage : count()
- Valeur minimale/maximale (tous types) : min(), max()
- Somme (de valeurs numériques) : sum()
- Statistiques : moyenne, médiane, écart-type (de valeurs numériques) : avg(), median(), variance(), stddev()

Résumé de l'enchaînement des étapes de l'exécution d'une requête agrégative

```
Select vendId, count(*) as cpt
  from dba.tbCommandes
where moisId=1
group by vendId
having count(*) < 3
order by vendId
```

1. From (et join) ➔ Utilisation de (**toutes** les lignes et colonnes de) **la table** spécifiée.
2. Where ➔ Eliminer des lignes → constitution d'un premier résultat temporaire.
3. Group by ➔ Partitionnement de ce résultat temporaire en fonction du critère spécifié.
4. Agrégation ➔ Sur ce partitionnement, évaluation de la fonction agrégative (ici **count**(*)) et constitution d'un nouveau résultat temporaire (1 ligne par critère).
5. Having ➔ Sur ce résultat, élimination des lignes qui ne satisfont pas la condition finale.
6. Order by ➔ Tri et affichage des colonnes reprises dans le **select.**

Quelques remarques :

1. Recourir aux alias, pour alléger.
2. Pas nécessairement **de fonction agrégative explicite dans select**.
3. Prédicat dans having :
   mêmes possibilités que dans where (and, or, not, between, in, …).
4. Questions complexes, cachant les détails de la réalisation :
   bien repérer les colonnes « identifiantes » (utiliser des alias).
5. A priori, impossibilité (?), d'obtenir une **synthèse par critère**, **et** une **synthèse générale en une seule** et même **requête**…
   Impossibilité ? FAUX ! → with rollup (dans le group by).

UPDATE…

Modification du contenu d'une table :

```
Update tableName
  set colName = expression [, …]
[ where predicat ]
```

- tableName : nom de **la** table dont le contenu est à modifier.
- Clause SET : permet de **spécifier la(les) colonne(s) dont le contenu est à modifier** ainsi que la valeur (littérale ou résultant d'une expression) à leur affecter (=).
  Les 2 pseudo-constantes DEFAULT et NULL sont **utilisables** comme expression.
- La clause WHERE correspond à la condition que doivent remplir les occurrences pour subir la mise à jour (filtre d'entrée).

Attention : si **la clause** WHERE ne figure pas **dans l'instruction**, toutes les occurrences (les lignes) **de la table** seront modifiées.

Modification et intégrité :

Pour modifier le contenu d'une table, il faut **disposer des droits** correspondants (grant update).
La modification d'une occurrence sera refusée si au moins une des contraintes d'intégrité est violée :

- **La valeur** spécifiée pour une colonne **n'appartient pas au domaine** de cette colonne
- **La valeur** spécifiée pour une colonne **rend fausse l'assertion CHECK** pour cette colonne
- La pseudo-constante **NULL** est **spécifiée pour une colonne** dont la définition comporte la clause **NOT NULL**, explicitement ou implicitement (clé candidate).

- Suite à la modification, la **valeur de la clé** primaire ou de toute autre clé candidate de l'occurrence **figure déjà dans** une **autre occurrence** (viol de l'unicité des clés candidates).
- La **valeur** spécifiée pour une colonne **clé étrangère n'a pas de correspondant** parmi les valeurs de la clé primaire de la table référencée ou n'est pas la pseudo-constante NULL.
- Si la table modifiée est référencée par une ou plusieurs contraintes d'intégrité référentielle, que la modification de l'occurrence comporte une **modification de la clé primaire et qu'au moins un chemin d'intégrité référentielle** partant de cette table **ne comporte pas la clause de mise à jour en CASCADE**.

DELETE FROM …

Suppression du contenu d'une table :

```
Delete from tableName
[ where predicat ]
```

- tableName: nom de la table dont le contenu est à supprimer en tout ou partie.
- La clause WHERE correspond à la condition que doivent remplir les occurrences (les lignes) pour subir la suppression (filtre d'entrée).

Attention : si **la clause** WHERE **ne figure** pas dans l'instruction, toutes les occurrences (les lignes) de la table seront supprimées. **Cela** ne **supprime** pas la table mais bien **TOUT son contenu** !

Suppression et intégrité :

Pour supprimer le contenu d'une table, il faut **disposer des droits** correspondants (grant delete).
La **suppression** d'une occurrence sera **refusée si** la **table** ainsi **modifiée** est **référencée par** une ou plusieurs **contraintes d'intégrité référentielle**, et qu'au moins **un chemin** d'intégrité référentielle partant de cette table ne comporte **pas la clause de suppression en CASCADE**.

Toute suppression acceptée entraîne **donc** la suppression **– au sein des tables référençantes –** des occurrences dont la clé étrangère possède la même valeur que la clé primaire de l'occurrence supprimée (suppression en cascade).
Donc**, risque de suppression en CASCADE si paramétrée, pouvant aller jusqu'à la suppression TOTALE de son contenu !**

INSERT INTO…

Ajout d'occurrences dans une table (valeurs littérales) :

```
Insert into tableName [(columnList)]
  values (valuesList) [, (…) ]
```

- La clause INTO spécifie le *nom de la table* dans laquelle on ajoute de nouvelles occurrences (lignes). Le nom de la table est éventuellement suivi des *noms de colonnes* de celle-ci ; cette liste n'est nécessaire que :
  - si l'ordre des valeurs (clause VALUES) ne correspond pas à l'ordre des colonnes de la table,
  - ou si on ne passe que les colonnes qui ont une valeur par défaut.
- La clause VALUES correspond à la *liste des valeurs* (littérales ou résultat d'une expression) constituant l'occurrence à insérer **dans le même ordre que la table ou de *columnList* si défini**.
  - pour toute colonne dont la définition comporte une valeur par défaut, celle-ci peut être insérée via la pseudo-constante DEFAULT,
  - de même pour toute colonne pour laquelle le *NULL* est autorisé, la pseudo-constante NULL peut être utilisée,
  - pour des colonnes de type 'temporel' (date, time, …), les constantes `current date`, `current time` … peuvent être utilisées.

Ajout d'occurrences dans une table (select) :

```
Insert into tableName (columnList)
  select …
    from
    [where …]
    [group by …]
    [having …]
```

On peut **insérer directement** des tuples (= lignes) **provenant d'une requête** qui renvoie autant de colonnes que nécessaire.

Ajout et intégrité :

Pour insérer des occurrences dans une table, **il faut** disposer **des droits** correspondants (grant insert). **L'ajout** d'une occurrence **sera refusée si** :

- la **valeur** spécifiée pour une colonne **n'appartient pas au domaine** de cette colonne,
- la **valeur** spécifiée pour une colonne **rend fausse l'assertion CHECK** pour cette colonne,
- la pseudo-constante **NULL** est **spécifiée pour une colonne** dont la définition comporte la clause **NOT NULL**, explicitement ou implicitement (clé candidate),
- la **valeur de la clé** primaire ou de toute autre clé candidate de l'occurrence à insérer figure **déjà dans une autre occurrence** (viol de l'unicité des clés candidates),
- la **valeur** spécifiée pour une colonne **clé étrangère n'a pas de correspondant** parmi les valeurs de la clé primaire de la table référencée **ou n'est pas** la pseudo-constante **NULL**.

Sous-requêtes

Une **sous-requête** (*requête imbriquée, requête en cascade*) **est** essentiellement **une requête** qui est rédigée et est exécutée **à l'intérieur d'une autre** requête (*requête principale*).

Il y a 3 types de sous-requêtes :

- Un **query « atomique »** ramenant **une seule valeur** qui sera utilisée dans les clauses conditionnelles (where, having) de la requête principale avec les **opérateurs « relationnels »**,
- Un **query « liste »** ramenant **plusieurs valeurs** qui seront utilisées dans les clauses conditionnelles (where, having) de la requête principale avec l'opérateur **IN**,
- Une **sous-requête « corrélée »** où la sous-requête dépend de la valeur d'un champ de l'occurrence (ligne) courante de la requête principale.

Une **requête « atomique »** ramène un résultat ne comportant qu'une seule colonne et une seule valeur (donc une seule ligne de résultat).
Typiquement : une requête agrégative mais sans partition (sans *group by*) portant sur toute une table, ou une restriction (clause *where* sur clé primaire).

Une **requête « liste »** **ramène** un résultat ne comportant **qu'une seule colonne mais plusieurs valeurs** (donc plusieurs lignes).

Les limites ?

Il n'y a **pas de limite quant au nombre de niveaux** de sous-requêtes (profondeur d'imbrication) **ou au nombre de sous-requêtes** (exemple : dans where et having).

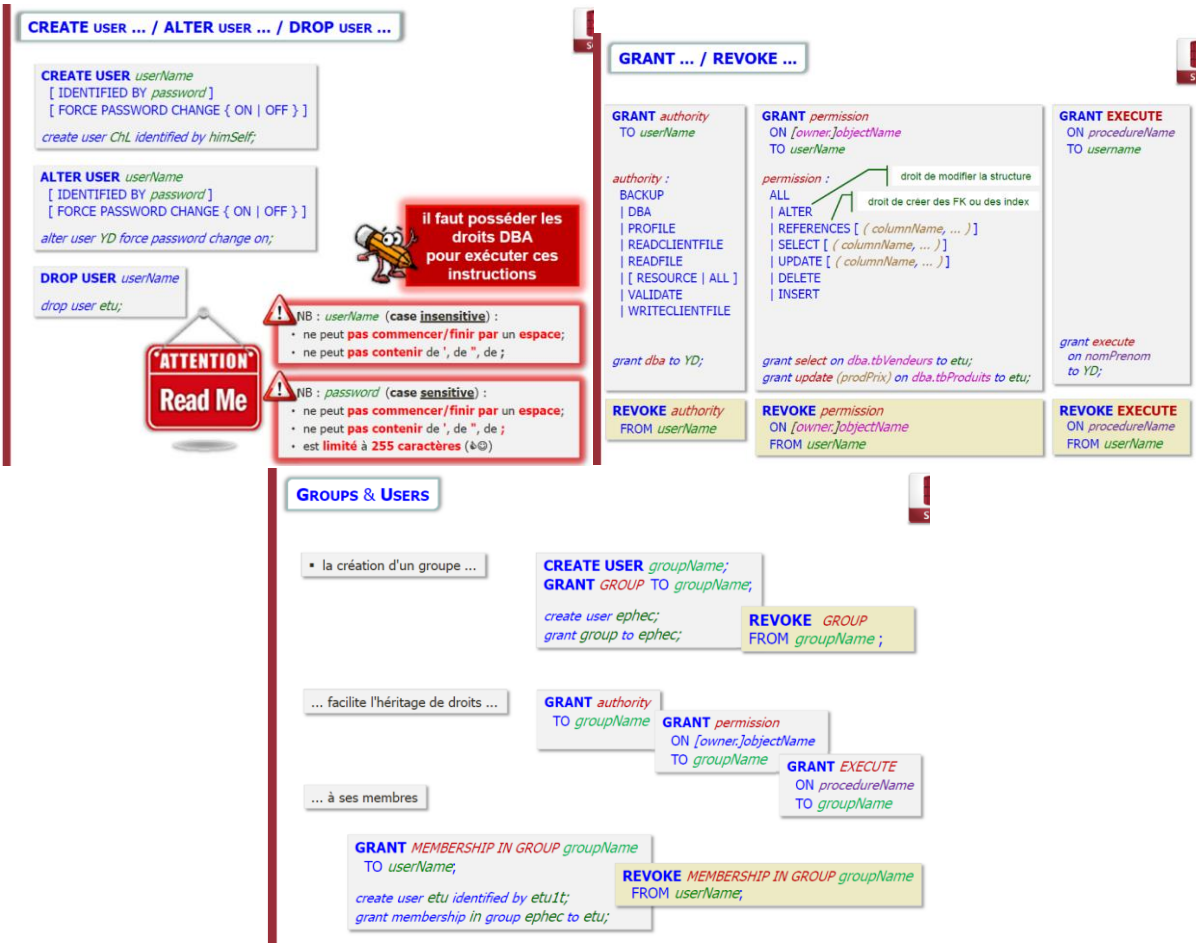
Où mettre une sous-requête ?

Partout où on peut mettre une valeur (scalaire), **on peut mettre une sous-requête atomique**… donc **y compris dans la clause select** elle-même.
Toute requête renvoie toujours un résultat sous forme d'une table dérivée (**même une requête**

atomique renvoie une table, d'une seule ligne et une seule colonne).
→ Partout où l'on peut mettre une table (c'est-à-dire essentiellement dans la clause *from*), on peut mettre une requête (**à condition de lui donner un nom via un alias** *as*).

Une sous-requête « corrélée » est une sous-requête qui dépend d'une valeur de la ligne courante de la requête principale ; concrètement, elle **comporte une référence à un champ de la requête principale via un alias** (variable de corrélation).
Une sous-requête corrélée est une excellente **alternative aux requêtes agrégatives avec partitionnement** (plus besoin de *group by… having…*). Une sous-requête corrélée permet d'**interroger** une base de données **sur** base de **l'abscence** (**l'inexistence**) de données. Pour ce faire, il faut utiliser un prédicat spécial : `where [not] exists (sub-query)`

Langage SQL : Data Control Language



Persistance des données

Un **SGBDR** (système de gestion de base de données relationnelle) traduit le « modèle relationnel » :

- par un ensemble nommé de **tables** (= relation) composées de **colonnes** (= **attribut**) et de **lignes** (= **tuple**)
- par une **contrainte d'unicité des lignes** (**primary key**)
- par la **définition pour chaque colonne**
  - de **domaines statiques** (éventuellement via des types conventionnels sur lesquels on précise une **contrainte**)
  - de **domaines dynamiques** (définis par la clé primaire d'une autre table) (**foreign key**)

Un SGBDR dispose pour son exploitation (au minimum) :

- d'un **langage non procédural** (4GL) **proche du langage naturel**, permettant :
  - la traduction du modèle et des contraintes (**DDL** : data definition language)
  - l'exploitation des données des tables à travers les noms et les valeurs (**DML** : data manipulation language)

Relations, attributs, tuples, tables, colonnes, lignes

Relations → Tables
Attributs → Colonnes
Tuples → Lignes

Pour garantir l'**unicité** des tuples, il est nécessaire de disposer d'un **attribut** (ou d'une composition d'attributs) **identifiant**(e) : c'est la **clé de la relation**.

Représentation d'une relation

Forme **tabulaire** :

- Nom de la relation → nom de la table (**table**)
- En-tête : noms des **attributs** → noms des colones ou champs (**columns**)
- Corps **de la relation** : tuples → lignes ou occurrences (**rows**)
- Attribut-**clé** : souligner le nom de la colonne (**primary key**)

Contraintes non-négociables :

- **Aucun ordre** imposé entre les **colonnes**
- **Aucun ordre** imposé entre les **lignes**
- **Unicité des occurrences**
- **Atomicité** (valeurs scalaires) **des valeurs des champs**
- **Manipulation : uniquement par les noms** (table, colonnes) **et les valeurs** (+ opérateurs de comparaison)

Le modèle relationnel

Extension de l'algèbre des **ensembles**

- Domaine : relations (disposant donc d'une clé identifiante)
- Opérateurs internes **ensemblistes** : (résultat = relation)
  - Intersection
  - Union
  - Différence
  - Produit cartésien
- Opérateurs **relationnels** spécifiques : (résultat = relation)
  - Projection
  - Restriction
  - Jointure
  - Division relationnelle

Union

Une relation R obtenue par l'UNION de 2 autres relations R1 et R2 est composée de toutes les occurrences des relations R1 et R2.

Attention au caractère nécessairement strictement relationnel du résultat : les lignes en double doivent donc être éliminées !

Intersection

Une relation R obtenue par l'intersection de 2 autres relations R1 et R2 est composée de toutes les occurrences communes aux relations R1 et R2. On la notera : $R = R1 \cap R2$.

Différence

Une relation R obtenue par la différence de 2 autres relations R1 et R2 est composée de toutes les occurrences de R1 ne figurant pas dans R2. On la notera : $R = R1 - R2$.

Produit cartésien

Une relation R obtenue par le produit cartésien de 2 autres relations R1 et R2 est composée de toutes les combinaisons possibles de chaque occurrence de R1 avec chaque occurrence de R2.
On la notera : $R = R1 \times R2$.

Projection

L'opération de projection d'une relation R1 construit une nouvelle relation R ne comportant qu'un sous-ensemble d'attributs (colonnes) de R1. On la notera : $R = [A]R1$

Restriction

La restriction sur une relation R1 construit une nouvelle relation R par extraction d'occurrences de R1 sur base d'un prédicat (condition) concernant des valeurs d'attributs. On la notera : $R = R1[P]$.

Jointure

La jointure de 2 relations R1 et R2 construit une relation résultante R par combinaison d'occurrences de R1 et de R2 en fonction du contenu d'un attribut commun. On la notera : $R = R1[A1 = A2]R2$.

Schéma normalisé

Schéma complet (explicite) :

- définit **chaque attribut** avec **son domaine/type** (*et d'autres contraintes*)
- définit les **champs identifiants** (les clés candidates) et sélectionne celui qui sera **clé primaire**
- **définit la ou les clés étrangères**

Modèle relationnel **conceptuel**

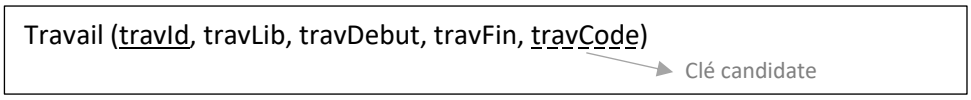
```
Travail (
  travId : numérique (entier non signé, numéro de suite), obligatoire
  travLib : chaîne de caractères (50 max), obligatoire
  travDebut : date (postérieure au 1/1/2017), obligatoire
  travFin : date (postérieure à travDebut), facultatif
  travCode : chaîne de caractères (5 max), obligatoire
)
CK : travId, CK : travCode, PK : travId
```

Modèle relationnel **physique**

```
CREATE TABLE tbTravaux (
  travId    INTEGER  NOT NULL DEFAULT AUTOINCREMENT,
  travLib   CHAR(50) NOT NULL,
  travDebut DATE     NOT NULL CHECK(@col > '2017-01-01'),
  travFin   DATE     DEFAULT NULL,
  travCode  CHAR(5)  NOT NULL,
  CONSTRAINT pkTrav PRIMARY KEY (travId ASC),
  CONSTRAINT ckTrav UNIQUE (travCode ASC),
  CONSTRAINT chkTravDates CHECK(travFin > travDebut)
)
```

Schéma simplifié

- Nom de relation, noms d'attributs, clé primaire soulignée



Buts des contraintes :

Une base de données dont le **contenu** (valeurs) est **cohérent**, garanti par le SGBDR

Le **respect** des **règles** d'**atomicité des attributs** et d'**unicité des occurrences** pour une relation la place en « première forme normale » (1NF).

Mais l'expression de toutes ces contraintes ne suffit (hélas) pas !
La **redondance** génère des anomalies de mémorisation lors des opérations de mise à jour (au sens large : ajout, modification, suppression).
C'est la traque de la redondance qui va justifier qu'un schéma relationnel contient plusieurs relations (qu'une base de données comporte plusieurs tables). Le processus d'**élimination de la redondance** et de ses conséquences néfastes porte le nom technique de **normalisation**. Il donne lieu à l'apparition d'une nouvelle contrainte de domaine : les clé étrangères (*foreign keys, FK*).

Un **domaine** défini par extension (liste des valeurs) peut toujours être (avantageusement) **remplacé par une table et une clé étrangère**.

JS – AJAX – JSON

Exportation depuis script JavaScript : JSON.stringify()
Importation vers script JavaScript : JSON.parse()

Voir syllabus JS – AJAX – JSON 02 pour :

- Affichage d'une table : génération thead et tbody (si colonnes connues ou non, méthode array/object/map)
- Tri (sort) de tableau
- Filtrage de tableau
- Tableaux associatifs : génération thead et tbody (si colonnes connues ou non, méthode array/object/map)

Syllabus JS – AJAX – JSON 03 :

Configuration d'un web server : (windows service)

```
-n srvWeb "D:\Web 2.0\Sites\Site_02\srvWeb"
-x tcpip
-xs http(port=99)
```

- -n : <nom du serveur> <chemin et nom de la db à démarrer>
- -x : <protocole de communication réseau> : -x tcpip
  - Le premier serveur (service) démarre automatiquement sur le port 2638. Les suivants reçoivent automatiquement un (autre) port. Pour spécifier manuellement : -x tcpip(port=…)
- -xs : <protocole de communication pour web services> : -xs http
  - Le port par défaut est 80. S'il est déjà pris, en **spécifier un explicitement** : -xs http(port=…). Le serveur TI est configuré sur le port 99.

Fonction getPath :

```
ALTER FUNCTION ‘’DBA’’.’’getPath’’()
returns long varchar
deterministic
BEGIN
  declare dbPath long varchar; //chemin de la db
  declare dbName long varchar; //nom de la db
  --
  set dbPath = (select db_property(‘file’)); -- path + nom de la db
  set dbName = (select db_property(‘name’)) + ‘.db’; -- nom de la db
  set dbPath = left(dbPath, length(dbPath) – length(dbName)); -- path seul
  --
  return dbPath; //renvoyer path
END
```

Pour renvoyer une page, on peut utiliser plusieurs méthodes :

- Fonction-système xp_read_file() de SA12 :
  xp_read_file system function : reads a file and returns the contents of the file as a LONG BINARY variable.
  Avec la fonction detPath() qu'on a créé, l'acquisition du chemin est dynamique.
  On crée une procédure http_getPage() :

```
ALTER PROCEDURE ‘’DBA’’.’’http_getPage’’(in url char(255))
result (html long varchar)
BEGIN
--
  call sa_set_http_header(‘Content-Type’, ‘text/html’); //header http
  select xp_read_file(dba.getPath() || url || ‘.html’); //renvoyer page
--
END
```

- Créer un web service http associé à une procédure stockée

```
CREATE SERVICE ‘’page’’
  TYPE ‘RAW’
  AUTHORIZATION OFF
  USER ‘’DBA’’
  URL ON
  METHODS ‘GET’
AS call dba.http_getPage(:url);
```

Procédures CSS, JS et IMG :

```
CREATE PROCEDURE DBA.http_getCSS(in url char(255))
result(css long varchar)
BEGIN
  call sa_set_http_header(‘Content-Type’, ‘text/css’); //header http
  select xp_read_file(dba.getPath() || ‘CSS\’ || url); //renvoyer css
END

CREATE PROCEDURE DBA.http_getJS(in url char(255))
result(js long varchar)
BEGIN
  call sa_set_http_header(‘Content-Type’, ‘text/javascript’); //header http
  select xp_read_file(dba.getPath() || ‘JS\’ || url); //renvoyer js
END
```

```
CREATE PROCEDURE DBA.http_getIMG(in url char(255))
result(img long binary)
BEGIN
  call sa_set_http_header('Content-Type', 'image/png'); //header http
  select xp_read_file(dba.getPath() || 'IMG\' || url); //renvoyer image
END
```

Services CSS, JS et IMG :

```
CREATE SERVICE "css" TYPE 'RAW'
  AUTHORIZATION OFF USER "DBA" URL ON
  METHODS 'GET'
AS call dba.http_getCSS(:url);
```

```
CREATE SERVICE "js" TYPE 'RAW'
  AUTHORIZATION OFF USER "DBA" URL ON
  METHODS 'GET'
AS call dba.http_getJS(:url);
```

```
CREATE SERVICE "img" TYPE 'RAW'
  AUTHORIZATION OFF USER "DBA" URL ON
  METHODS 'GET'
AS call dba.http_getIMG(:url);
```

Service "transparent" :

```
CREATE SERVICE "root" TYPE 'RAW'
  AUTHORIZATION OFF USER "DBA" URL ON
  METHODS 'GET'
AS call dba.http_getPage(:url);
```

AJAX : Asynchronous JAvascript and Xml

(voir syllabus JS – AJAX – JSON 04)

Architecture utilisant de manière conjointe diverses technologies normalisées disponibles sur les navigateurs pour réaliser des applications client/serveur web dynamiques.

- DOM : modèle objet (programmable) de la page web (**1998**)
- JAVASCRIPT : langage de programmation événementiel
- HTML5/CSS3 : structure et mise en forme de la page
- XML/JSON : format normalisé d'échange de données (**2002 JSON**)
- **Librairie/objet JS spécifique** : **XMLHttpRequest** : permet d'effectuer des requêtes http vers un serveur et de recevoir la réponse (**2002**)

Asynchronous JavaScript

Ensemble de technologies de type web (base protocole http) : **DOM + CSS + JAVASCRIPT + XMLHttpRequest**

Méthodes :
instanciation : new *//var xhr = new XMLHttpRequest()*
.open(httpMethod, [service+]url) //xhr.open('GET', 'img\lorem.jpg')
.send() //xhr.send()
.abort()
.setRequestHeader()
.getResponseHeader()
.getAllResponseHeaders()

Propriétés :
.status //code http (200, 404, …)
.statusText //message http
.readyState //état connexion
.responseText //data reçues du serveur

readyState:
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request **finished** and **response is ready**

Evénement(s) :
.onreadystatechange

AJAX est asynchrone (notion de « promesse »). La réponse est reçue de manière **événementielle différée** : `.onreadystatechange = callbackFunction`.
Donc, les informations attendues par le client ne lui sont pas forcément fournies dans le même ordre que les demandes.

Fonction xhrReqHtml :

Une fonction **réutilisable** (**paramétrée**) avec une **variable locale** et une fonction **callback anonyme**.

```
function xhrReqHtml(url, id){
  //place dans l'élément html d'identifiant id
  //la réponse de la requête ajax demandant la ressource url
  var xhr = new XMLHttpRequest(); //instancier XMLHttpRequest
  xhr.open('get', url, true) ;         //préparer
  xhr.onreadystatechange =             //callback : fonction anonyme
    function(){
      if (xhr.status == 200 && xhr.readyState == 4) {
        setElem(id, xhr.responseText);
      }
    }
  xhr.send()              //envoyer
}

function showBox(n) {
  xhrReqHtml('lorem' + n, 'ipsum')     //<p id=ipsum>
}
```

AJAX + JSON

Exemple de requête transformé en procédure JSON :

```
CREATE PROCEDURE dba.getLangues()
RESULT(id char(2), lib char(30)) //noms en sortie
BEGIN
  //header http
  call sa_set_http_header('Content-Type', 'application:json; charset=utf-8');
  //query
  select langId, langLib
    from dba.tbLangues
END
```

Web Service de type JSON qui invoque la procédure ci-dessus :

```
CREATE SERVICE ''getLangues''
  TYPE 'JSON'
  AUTHORIZATION OFF
  USER ''DBA''
  URL ON
  METHODS 'GET'
AS call dba.getLangues();
```

Invoquer le service en JavaScript/AJAX :

```
<script>
  var xhr = new XMLHttpRequest();
  function initPage(){
    xhr.open('get', 'getLangues', true);
    …
  }
</script>
```

Parser une réponse en JSON pour récupérer un Array :

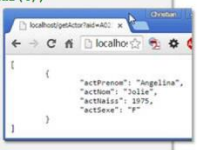
```
…
var lng = JSON.parse(xhr.responseText);
…
```

SQL - Web Services – AJAX : pages web & requêtes « atomiques »

Différentes grandes possibilités pour ramener du contenu vers la page (3)

1. Un query alimentant un web service 'JSON' (serveur « léger » / client « **lourd** ») :
   - **côté serveur** : le query est dans une procédure qui renvoie un 'recordset' (tableau relationnel) classique, linéarisé par le web service de type JSON

```
CREATE PROCEDURE "DBA"."sp_getActor"(in aid char(4))
BEGIN
  select actPrenom, actNom, actNaiss, actSexe
    from tbActeurs
   where actId = aid
END;
CREATE SERVICE "getActor" TYPE 'JSON'
  AUTHORIZATION OFF USER "DBA"
  URL ON METHODS 'GET'
  AS call sp_getActor(:aid);
```



   - **côté client**, la chaîne doit d'abord être acquise par une requête AJAX, puis parsée pour obtenir un Array qui est enfin **itéré (for)** pour fabriquer des éléments HTML

```
xhr.open('get', 'getActor?aid=A023', true);
xhr.onload = function(){
  var arr = JSON.parse(xhr.responseText); var html = '';
  for(var el in arr[0]){
    html += '<div id=' + el + '>' + arr[0][el] + '</div>';
  }
  // ...
}
```

SQL - Web Services – AJAX : pages web & requêtes « atomiques »

Différentes grandes possibilités pour ramener du contenu vers la page (4)

2. Un query alimentant un web service « RAW » (serveur « **lourd** », client « léger ») :
   - **côté serveur** : le query doit lui-même doit fabriquer un texte au format HTML (par un query atomique); celui-ci est envoyé tel quel par le web service

```
CREATE PROCEDURE "DBA"."sp_getActor"(in aid char(4))
BEGIN
  declare html long varchar; // texte à générer
  set html = (select
      string('<div id=prn>', actPrenom, '</div>') ||
      string('<div id=nm>', actNom, '</div>') ||
      string('<div id=nai>', actNaiss, '</div>') ||
      string('<div id=sex>', actSexe, '</div>')
    from tbActeurs
   where actId = aid);
  select html; // renvoi texte
END;
CREATE SERVICE "getActor" TYPE 'RAW'
  AUTHORIZATION OFF USER "DBA" URL ON METHODS 'GET'
  AS call sp_getActor(:aid);
```

   - **côté client**, le texte HTML doit d'abord être acquis par une requête AJAX puis envoyé directement - tel qu'- dans la propriété innerHTML de l'élément-cible de la page

```
xhr.open('get', 'getActor?aid=A023', true);
xhr.onload = function(){setElem('act', xhr.responseText;}
```

Deux syntaxes pour rédiger un query atomique pour générer du texte

```
CREATE PROCEDURE "DBA"."sp_getActor"(in aid char(4))
BEGIN
  declare html long varchar;
  set html = (select
      string('<div id=prn>', actPrenom, '</div>') ||
      string('<div id=nm>', actNom, '</div>') ||
      string('<div id=nai>', actNaiss, '</div>') ||
      string('<div id=sex>', actSexe, '</div>')
    from tbActeurs
   where actId = aid);
  select html; // renvoi texte
END;
```

```
CREATE PROCEDURE "DBA"."sp_getActor"(in aid char(4))
BEGIN
  declare html long varchar;
  select
      string('<div id=prn>', actPrenom, '</div>') ||
      string('<div id=nm>', actNom, '</div>') ||
      string('<div id=nai>', actNaiss, '</div>') ||
      string('<div id=sex>', actSexe, '</div>') into html
    from tbActeurs
   where actId = aid;
  select html; // renvoi texte
END;
```

atomisation de requêtes-listes : la fonction agrégative `list()` (1)

problème : les champs d'un select sont tous de nature atomique (valeur scalaire) et on ne peut pas y placer une sous-requête renvoyant un tableau

quels sont les titres des films où joue l'acteur A023 ?

```
select filmTitre
  from tbRoles natural join tbFilms
 where actId = 'A023'
```

Résultats

filmTitre
1 By the sea
2 Tomb Raider
3 Mr. & Mrs Smith

donc ceci est impossible :

```
select actPrenom, actNom, actNaiss, actSexe,
       (select filmTitre
          from tbRoles natural join tbFilms
         where actId = 'A023') as films
  from tbActeurs
 where actId = 'A023'
```

pour obtenir cela dans une page web, il faudrait donc multiplier les requêtes (une pour la fiche de l'acteur, une pour la liste de ses films), les services web, les requêtes Ajax, les fonctions de formatage JavaScript ... ➔ donc privilégier la solutions « **client lourd** » ...

... sauf que ...

atomisation de requêtes-listes : la fonction agrégative `list()` (2)

la fonction linéaire un tableau de valeurs sous forme d'une chaîne par concaténation

quels sont les titres des films où joue l'acteur A023 ?

```
select list(filmTitre) as films
  from tbRoles natural join tbFilms
 where actId = 'A023'
```

Résultats

films
1 By the sea,Tomb Raider,Mr. & Mrs Smith

par défaut, le séparateur est la virgule (sans espace)
mais on peut choisir son propre séparateur (chaîne) :

```
select list(filmTitre, ', ') as films
  from tbRoles natural join tbFilms
 where actId = 'A023'
```

on a donc une solution pour renvoyer un « **sous**-tableau »

```
select actPrenom, actNom, actNaiss, actSexe,
       (select list(filmTitre, ', ')
          from tbRoles natural join tbFilms
         where actId = 'A023') as films
  from tbActeurs
 where actId = 'A023'
```

Résultats

actPrenom	actNom	actNaiss	actSexe	films
1 Angelina	Jolie	1,975	F	By the sea, Tomb Raider, Mr. & Mrs Smith

atomisation de requêtes-listes : solution serveur « **lourd** » / client « léger »

la fonction list() permet même de générer de « vrais » tableaux html côté serveur

```
CREATE PROCEDURE "DBA"."sp_getActorFilms"(in aid char(4))
BEGIN
  declare act long varchar;
  declare flm long varchar;
  set act = (select
      string('<div id=prn>', actPrenom, '</div>') ||
      string('<div id=nm>', actNom, '</div>') ||
      string('<div id=nai>', actNaiss, '</div>') ||
      string('<div id=sex>', actSexe, '</div>')
    from tbActeurs where actId = aid);
  set flm = string(
    '<div id=flm>',
    (select string('<table><tbody><tr>',
                   list(string('<td>', filmTitre, '</td>'), '</tr><tr>'),
                   '</tr></tbody></table>')
       from tbRoles natural join tbFilms
      where actId = aid),
    '</div>');
  select string(act, flm); // renvoi texte
END;
```

(raw)

atomisation de requêtes-listes : solution serveur « **lourd** » / client « **lourd** »

```
CREATE PROCEDURE "DBA"."sp_getActor"(in aid char(4))
BEGIN
  select actPrenom, actNom, actNaiss, actSexe,
         (select list(filmTitre, ', ')
            from tbRoles natural join tbFilms
           where actId = aid) as films
    from tbActeurs where actId = aid
END;
```

(json)

atomisation de requêtes-listes : solution serveur « **lourd** » / client « léger »

```
CREATE PROCEDURE "DBA"."sp_getActorFilms"(in aid char(4))
BEGIN
  declare act long varchar;
  declare flm long varchar;
  set act = (select
      string('<div id=prn>', actPrenom, '</div>') ||
      string('<div id=nm>', actNom, '</div>') ||
      string('<div id=nai>', actNaiss, '</div>') ||
      string('<div id=sex>', actSexe, '</div>')
    from tbActeurs where actId = aid);
  set flm = string('<div id=flm>',
                   (select list(filmTitre, ', ')
                      from tbRoles natural join tbFilms
                     where actId = aid),
                   '</div>');
  select string(act, flm); // renvoi texte
END;
```

(raw)