

Les règles de Codd

Pour les bases de données relationnelles



par Frédéric Brouard, alias SQLpro
MVP SQL Server
Expert langage SQL, SGBDR, modélisation de données

Auteur de :

- SQLpro <http://sqlpro.developpez.com/>
 - "SQL", coll. Synthex, avec C. Soutou, Pearson Education 2005
 - "SQL" coll. Développement, Campus Press 2001
- Enseignant aux Arts & Métiers et à l'ISEN Toulon

Les douze règles de Codd (numérotés de 0 à 12, donc en fait treize...) formalisent la notion de SGBD relationnel. C'est bien le docteur Codd, père fondateur de l'algèbre relationnelle qui grave ainsi dans le marbre les 13 commandements de la Loi pour tout SGBD se prétendant relationnel. Ces règles sont basées sur ses travaux originaux effectués à partir de 1970, et ont été publiées dans deux articles de vulgarisation du magazine Computerworld (octobre 1985) : Is your DBMS really relational ? et Does your DBMS run by the rules?

Ces règles constituent les fondements des bases de données relationnelles et permettent de connaître le niveau relationnel du produit d'un éditeur. Ne pas s'y contraindre impliquera plus d'inconvénients que d'avantages.

En un sens elles gravent dans le marbre "la Loi" que tout bon SGBD relationnel devrait suivre.

Juridiquement il s'agit là du document le plus précieux concernant les règles de l'art des bases de données relationnelles.

Copyright et droits d'auteurs : La Loi du 11 mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part que *des copies ou reproductions strictement réservées à l'usage privé et non [...] à une utilisation collective*, et d'autre part que les analyses et courtes citations dans un but d'illustration, toute reproduction intégrale ou partielle faite sans le consentement de l'auteur [...] est illicite. Le présent article étant la propriété intellectuelle de Frédéric Brouard, prière de contacter l'auteur pour toute demande d'utilisation, autre que prévu par la Loi à SQLpro@SQLspot.com

0 - Les règles brutes :

RÈGLE 0 – Introduction :

Tout système de bases de données prétendant être relationnel doit être capable de gérer complètement les bases de données à l'aide de ses caractéristiques relationnelles.

RÈGLE 1 - Règle de l'information :

Toutes les informations dans une base de données relationnelle sont représentées de façon explicite au niveau logique et d'une seule manière : par des valeurs dans des tables.

RÈGLE 2 - Garantie d'accès :

Toute donnée atomique d'une base de données relationnelle est accessible par le biais d'une combinaison du nom de la table ou de la vue, d'une valeur de la clé primaire et d'un nom de colonne.

RÈGLE 3 - Gestion du marqueur NULL :

Les marqueurs NULL sont systématiquement pris en charge pour représenter des informations manquantes ou inapplicables, indépendamment du type, du domaine de valeur ou d'une valeur par défaut.

RÈGLE 4 - Catalogue relationnel, dynamique et accessible directement :

La description de la base de données et de son contenu est représentée au niveau logique de la même manière que les données ordinaires (des tables).

RÈGLE 5 - Langage de manipulation de données complet :

Au moins un des langages du SGBDR doit avoir une syntaxe complète et doit permettre la définition des données, la formation des vues, la manipulation des données, la gestion des règles d'intégrité, les autorisations et les frontières des transactions.

RÈGLE 6 - Règle de mise à jour des vues :

Toutes les vues qui sont théoriquement modifiables peuvent être mises à jour par le système.

RÈGLE 7 - Insertion, suppression et modification ensemblistes :

Le SGBDR retourne un ensemble d'éléments en réponse aux requêtes qui lui sont soumises. Il doit pouvoir mettre à jour un ensemble d'éléments en exécutant une seule requête.

RÈGLE 8 - Indépendance physique des données :

Les applications et les programmes terminaux sont logiquement in affectés lorsque les méthodes d'accès physiques ou les structures de stockage sont modifiées.

RÈGLE 9 - Indépendance logique des données :

Les applications et les programmes terminaux sont logiquement in affectés, quand des changements de tous ordres, préservant les informations et qui ne leur portent théoriquement aucune atteinte, sont apportés aux tables de base (restructuration).

RÈGLE 10 - Indépendance vis-à-vis de l'intégrité :

Les contraintes d'intégrité spécifiques à une base de données relationnelle sont définies à l'aide du langage relationnel et leur définition doit être stockée dans le catalogue et non dans des programmes d'application.

RÈGLE 11 - Indépendance de distribution :

Le langage relationnel doit permettre aux programmes d'application et aux requêtes de demeurer identiques sur le plan logique lorsque des données, quelles qu'elles soient, sont physiquement réparties ou centralisées.

RÈGLE 12 - Non subversion :

Il ne doit pas être possible de transgresser les règles d'intégrité et les contraintes définies par le langage relationnel du SGBDR en utilisant un langage de plus bas niveau (gérant une seule ligne à la fois).

1 - Explications

Nous les présentons maintenant par domaine fonctionnel en les commentant...

1.0 - Règles de fondement

Un produit ne peut être considéré comme SGBDR que s'il respecte impérativement ces règles.
Règles concernées : 0 et 12.

RÈGLE 0 – Introduction :

Tout système de bases de données prétendant être relationnel doit être capable de gérer complètement les bases de données à l'aide de ses caractéristiques relationnelles.

Le SGBDR ne peut faire appel à aucune opération non relationnelle pour réaliser la moindre de ses fonctionnalités. Par exemple un SGBDR qui demanderait à lire tel ou tel fichier pour donner des informations sur la structure d'une base de données, ou bien qui imposerait de spécifier des caractéristiques technique (donc non logique) pour définir les objets de la base, ne peut prétendre à être relationnel. On en reviendrait alors au temps du Cobol qui mélangeait

allégrement les problématiques physiques tel que les chemins d'accès aux fichiers, les formats des données.... ce qui obligeait à produire un code non portable !

DISCUSSION : par exemple, sur ce point, on peut se demander si MySQL qui oblige à indiquer la nature physique de la table (moteur INNO db, MyIsam...) à chaque création, ne viole pas cette règle...

RÈGLE 12 - Non subversion :

Il ne doit pas être possible de transgresser les règles d'intégrité et les contraintes définies par le langage relationnel du SGBDR en utilisant un langage de plus bas niveau (gérant une seule ligne à la fois).

Tous les accès aux données de la base se font sous le contrôle du SGBDR, même s'il y a traitement d'une seule ligne par langage client.

DISCUSSION : arrêtons-nous un instant sur un exemple. Soit la table TAB composée d'une seule colonne COL, dotée d'une contrainte d'unicité (UNIQUE) qui empêche de saisir deux lignes ayant les mêmes données :

```
CREATE TABLE TAB (COL INT UNIQUE);
INSERT INTO TAB (COL) VALUES (1);
INSERT INTO TAB (COL) VALUES (2);
INSERT INTO TAB (COL) VALUES (3);
```

Que va t-il se passer si nous voulons incrémenter toutes ces valeurs d'une unité ?

Si cette modification est entreprise à l'aide d'un accès via un langage de bas niveau manipulant les données ligne par ligne, elle a de fortes chances d'échouer, du fait de la contrainte d'unicité de la colonne.

En revanche, l'application de la commande SQL : ...

```
UPDATE TAB SET COL = COL + 1;
```

...réussira, car SQL est par nature ensembliste et de ce fait ne vérifiera l'unicité de la colonne qu'après avoir mis à jour toutes les lignes impactées (voir règle 7).

1.1 - Règles structurelles

Codd introduit la notion fondamentale de relation (c'est à dire le contenant des données). Cela induit les éléments suivants : domaine, clé primaire, clé étrangère, etc.

Règles concernées : 1 et 6.

RÈGLE 1 - Règle de l'information :

Toutes les informations dans une base de données relationnelle sont représentées de façon explicite au niveau logique et d'une seule manière : par des valeurs dans des tables.

Toutes les informations d'une base sont stockées dans des relations et gérées de manière uniforme et unique, la table (des colonnes, des lignes, des valeurs) et ses corollaires : clef, domaine...

Autrement dit, aucune donnée d'exploitation ne doit figurer en dehors d'un accès à la table.

DISCUSSION : dès lors on doit se poser des questions comme, que faire des données unitaires de l'application sous jacente à la base : par exemple si l'on veut indiquer la date de création de la base, ses auteurs, ses droits juridiques, divers paramètres spécifiques ? La réponse est alors assez simple : il suffit de créer une table de paramétrage. Peu importe si cette table ne contient qu'un nombre infime de lignes.

Dans ce cas, deux modèles s'affrontent : la table mono ligne ou la table multi ligne.

Table mono ligne :

```
CREATE TABLE T_PARAM_PRM
(PRM_NOM_APPLICATION VARCHAR(32),
 PRM_DATE_CREATION DATE,
 PRM_AUTEUR VARCHAR(256),
 PRM_COPYRIGHT VARCHAR(512)
...
)
```

Inconvénient : nécessite de restructurer la table pour tout ajout de nouveau paramètre et de plus n'est pas indexable

Table multi ligne :

```
CREATE TABLE T_PARAM_PRM
(PRM_NOM_PARAMETRE VARCHAR(32),
 PRM_TYPE_PARAMETRE VARCHAR(8),
 PRM_VALEUR_PARAMETRE VARCHAR(1024)
)
```

Avantages : pas de restructuration pour ajouter un nouveau paramètre, table indexable...

RÈGLE 6 - Règle de mise à jour des vues :

Toutes les vues qui sont théoriquement modifiables peuvent être mises à jour par le système.

Si une vue peut théoriquement être mise à jour, c'est à dire que l'on peut insérer ou supprimer des lignes ou encore modifier la valeur de certaines données, alors le SGBDR doit pouvoir effectuer cette opération. Cette particularité est souvent hélas ignorée des développeurs. Et l'intérêt des vues, comme leur importance est mal comprise. **En fait on ne devrait jamais passer que par des vues pour accéder aux données.** Toute application ne devrait jamais manipuler les données que par des vues et par extension, des procédures stockées. L'utilisation systématique de vues comme objet d'accès aux données constitue la fameuse "vue externe" des 4 vues du schéma relationnel (*conceptuel, logique, physique, externe*).

DISCUSSION : Malheureusement aucun SGBDR actuel ne respecte cette règle à la lettre. En effet, les conditions mathématiques de faisabilité n'ont pas encore été toutes découvertes (forte complexité) pour identifier les vues théoriquement susceptibles d'être mis à jour. La plupart des éditeurs se limitent donc à proposer des vues à travers lesquelles les mises à jour de données sont possibles à condition qu'elles présentent des données même des plusieurs tables, mais avec les clefs (primaire ou unique) et sans aucune transformation ou ajout de données. Cependant les SGBDR les plus sophistiqués mettent en oeuvre des techniques capables de simuler la mise à jour dans des vues complexes faisant appel à de nombreuses tables, par le biais de trigger «°INSTEAD OF°»...

Exemple :

```
-- partant d'une table des clients
CREATE TABLE T_CLIENT_CLI
(CLI_ID INT NOT NULL PRIMARY KEY,
```

```

    CLI_NOM      CHAR(32)      NOT NULL,
    CLI_PRENOM   VARCHAR(25)
);

-- créons la vue présentant les clients dont les noms vont de A à M :
CREATE VIEW V_CLIENT_AM
AS
SELECT CLI_ID, CLI_NOM, CLI_PRENOM
FROM   T_CLIENT_CLI
WHERE  CLI_NOM < 'N';

-- puis une autre vue présentant les clients dont les noms vont de N à Z :
CREATE VIEW V_CLIENT_NZ
AS
SELECT CLI_ID, CLI_NOM, CLI_PRENOM
FROM   T_CLIENT_CLI
WHERE  CLI_NOM >= 'N';

-- Il n'y a aucune transformation de données et les vues ne portent
-- que sur une table. Les vues comportent aussi la clef...

-- Est-il possible d'insérer une donnée dans l'une des vues,
-- même si cette données ne respecte pas le filtre WHERE ?
INSERT INTO V_CLIENT_AM
VALUES (1, 'ZARDOZ', 'marcel');

-- c'est réussi !

SELECT *
FROM   V_CLIENT_AM;

CLI_ID      CLI_NOM      CLI_PRENOM
-----
1           ZARDOZ      marcel

SELECT *
FROM   V_CLIENT_NZ;

CLI_ID      CLI_NOM      CLI_PRENOM
-----
1           ZARDOZ      marcel

-- La modification dans la vue est-elle possible ?
UPDATE V_CLIENT_AM
SET     CLI_NOM = REPLACE(CLI_NOM, 'Z', 'A')
-- Elle n'a aucun effet sur ZARDOZ ! Nous ne pouvons le modifier par ce biais.

-- modifions par l'intermédiaire de l'autre vue :
UPDATE V_CLIENT_NZ
SET     CLI_NOM = REPLACE(CLI_NOM, 'Z', 'A')

(1 ligne(s) affectée(s))

-- c'est réussit :
SELECT *
FROM   V_CLIENT_AM;

CLI_ID      CLI_NOM      CLI_PRENOM
-----
1           AARDOA      marcel

-- peut-on supprimer un client qui n'est pas visible dans la vue
-- par l'intermédiaire de cette vue ?
DELETE FROM V_CLIENT_NZ
WHERE  CLI_NOM = 'AARDOA'

(0 ligne(s) affectée(s))

```

Moralité : on peut insérer de nouvelles lignes dans une vue, même si les données de la vue ne respectent pas le filtre WHERE ! Mais ceci peut être modifié avec la clause WITH CHECK OPTION, qui oblige les mises à jour à respecter les filtres de vue. En revanche la suppression de lignes comme la modification des valeurs de certaines colonnes, ne peut se faire que sur les lignes présentées par la vue...

Autre exemple :

```
-- créons une table simplissime d'une seule colonne de type entier
CREATE TABLE T_INT (I INT);

-- créons une vue non moins simplissime d'une seule colonne calculé
CREATE VIEW V_INT
AS
SELECT I + 1 AS J
FROM T_INT;
-- juste une petite soustraction d'une unité...

-- comme vous pourrez le constater, toute mise à jour (INSERT, UPDATE, DELETE)
-- dans cette vue est impossible...
INSERT INTO V_INT VALUES (0)

Erreur : Impossible de mettre à jour ou d'insérer une vue ou une fonction 'V_INT'
car elles contiennent un champ d'une dérivée ou d'une constante.
```

Quelle est la raison de ce message d'erreur ? Tout simplement la plage de validité des données n'étant plus la même, SQL, par sécurité, refuse que les données et les lignes d'une telle vue soient modifiées... pourtant la transformation inverse est d'une grande simplicité. Il n'en aurait pas été de même si nous avions appliqué une fonction LOG ou SINUS !

Troisième exemple :

```
-- à partir des tables précédente, créons une vue avec une jointure...
CREATE VIEW V
AS
SELECT *
FROM T_CLIENT_CLI AS C
INNER JOIN T_INT AS I
ON C.CLI_ID = I.I

-- peut-on y insérer une ligne ?
INSERT INTO V (CLI_ID, CLI_NOM, CLI_PRENOM) VALUES (99, 'DURAND', 'Paul')
```

Oui il est possible d'insérer des lignes dans une vue comportant des jointures de différentes tables, à condition que l'insertion ne porte que sur l'une des tables... Bien entendu il ne faut pas qu'il y ait de transformation de données...

Dernier exemple

Voici comment en utilisant les techniques du mapping relationnel objet on peut insérer des données à travers une vue...

Le principe consiste à créer un objet composé de plusieurs tables. Ici une personne avec deux adresses, l'une de livraison, l'autre de facturation. Pour cela on dispose de tables de référence (type d'adresse, civilité...) et de tables d'entité (personne, adresse, ligne d'adresse).

```
/*
-- CRÉATION DES TABLES
-- création d'une table de référence pour les types d'adresse :
CREATE TABLE T_TYPE_ADRESSE_TAD
(TAD_ID INT NOT NULL PRIMARY KEY,
```

```
TAD_LIBELLE          CHAR(16) NOT NULL)
GO

-- insertion des références pour les adresses :
INSERT INTO T_TYPE_ADRESSE_TAD VALUES (1, 'LIVRAISON')
INSERT INTO T_TYPE_ADRESSE_TAD VALUES (2, 'FACTURATION')
GO

-- création d'une table de référence pour les titres de personnes (civilité) :
CREATE TABLE T_TITRE_TTR
(TTR_ID              INT NOT NULL PRIMARY KEY,
 TTR_LIBELLE        CHAR(16) NOT NULL,
 TTR_ABREGE         CHAR(5) NOT NULL)
GO

-- insertion des références pour les titres :
INSERT INTO T_TITRE_TTR VALUES (1, 'Monsieur', 'M.')
INSERT INTO T_TITRE_TTR VALUES (2, 'Madame', 'Mme.')
INSERT INTO T_TITRE_TTR VALUES (3, 'Mademoiselle', 'Mlle.')
GO

-- création d'une table de personnes
CREATE TABLE T_PERSONNE_PRS
(PRS_ID              INT NOT NULL IDENTITY PRIMARY KEY,
 PRS_NOM             CHAR(32) NOT NULL,
 PRS_PRENOM         VARCHAR(25),
 TTR_ID              INT FOREIGN KEY REFERENCES T_TITRE_TTR (TTR_ID))
GO

-- création d'une table d'adresse
CREATE TABLE T_ADRESSE_ADR
(ADR_ID              INT NOT NULL IDENTITY PRIMARY KEY,
 PRS_ID              INT NOT NULL FOREIGN KEY REFERENCES T_PERSONNE_PRS (PRS_ID),
 TAD_ID              INT NOT NULL FOREIGN KEY REFERENCES T_TYPE_ADRESSE_TAD(TAD_ID),
 ADR_CP              CHAR(5) NOT NULL,
 ADR_VILLE           CHAR(32) NOT NULL,
 CONSTRAINT UK_PRSTAD UNIQUE (PRS_ID, TAD_ID))
GO

-- création d'une table de lignes d'adresse.
-- Notez que l'on a contraint à 4 lignes par adresse au maximum...
CREATE TABLE T_LIGNE_ADRESSE_LAD
(LAD_ID              INT NOT NULL IDENTITY PRIMARY KEY,
 ADR_ID              INT NOT NULL FOREIGN KEY REFERENCES T_ADRESSE_ADR (ADR_ID),
 LAD_POSITION        SMALLINT NOT NULL CHECK (LAD_POSITION BETWEEN 1 AND 4),
 LAD_LIGNE           VARCHAR(38) NOT NULL,
 CONSTRAINT UK_POSLAD UNIQUE (ADR_ID, LAD_POSITION))
GO

/*****
-- CRÉATION DE LA VUE
*****/
-- vue composite (11 jointures...) pour mise à plat des informations de personne
CREATE VIEW V_PERSONNE_ADRESSE_PAD
AS
SELECT PRS.PRS_ID, PRS_NOM, PRS_PRENOM,
       TTR.TTR_ID, TTR_LIBELLE, TTR_ABREGE,
       ADL.ADR_ID AS ADR_ID_LIVRAISON,
       ADL.ADR_CP AS ADR_CP_LIVRAISON,
       ADL.ADR_VILLE AS ADR_VILLE_LIVRAISON,
       AL1.LAD_LIGNE AS LAD_LIGNE_LIVRAISON_1,
       AL2.LAD_LIGNE AS LAD_LIGNE_LIVRAISON_2,
       AL3.LAD_LIGNE AS LAD_LIGNE_LIVRAISON_3,
       AL4.LAD_LIGNE AS LAD_LIGNE_LIVRAISON_4,
       ADF.ADR_ID AS ADR_ID_FACTURATION,
       ADF.ADR_CP AS ADR_CP_FACTURATION,
       ADF.ADR_VILLE AS ADR_VILLE_FACTURATION,
       AF1.LAD_LIGNE AS LAD_LIGNE_FACTURATION_1,
```



```

        AF2.LAD_LIGNE AS LAD_LIGNE_FACTURATION_2,
        AF3.LAD_LIGNE AS LAD_LIGNE_FACTURATION_3,
        AF4.LAD_LIGNE AS LAD_LIGNE_FACTURATION_4
FROM    T_PERSONNE_PRS AS PRS
-- jointure pour titre
        LEFT OUTER JOIN T_TITRE_TTR AS TTR
            ON PRS.TTR_ID = TTR.TTR_ID
-- jointure pour adresse livraison
        LEFT OUTER JOIN T_ADRESSE_ADR AS ADL
            ON      ADL.PRS_ID = PRS.PRS_ID
               AND ADL.TAD_ID = (SELECT TAD_ID
                                   FROM   T_TYPE_ADRESSE_TAD
                                   WHERE  TAD_LIBELLE = 'LIVRAISON'
                                   COLLATE French_CI_AI)
-- jointures pour lignes d'adresse de livraison :
-- ligne 1 pour adresse livraison
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AL1
            ON ADL.ADR_ID = AL1.ADR_ID AND AL1.LAD_POSITION = 1
-- ligne 2 pour adresse livraison
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AL2
            ON ADL.ADR_ID = AL2.ADR_ID AND AL2.LAD_POSITION = 2
-- ligne 3 pour adresse livraison
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AL3
            ON ADL.ADR_ID = AL3.ADR_ID AND AL3.LAD_POSITION = 3
-- ligne 4 pour adresse livraison
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AL4
            ON ADL.ADR_ID = AL4.ADR_ID AND AL4.LAD_POSITION = 4
-- jointure pour adresse facturation
        LEFT OUTER JOIN T_ADRESSE_ADR AS ADF
            ON      ADF.PRS_ID = PRS.PRS_ID
               AND ADF.TAD_ID = (SELECT TAD_ID
                                   FROM   T_TYPE_ADRESSE_TAD
                                   WHERE  TAD_LIBELLE = 'FACTURATION'
                                   COLLATE French_CI_AI)
-- jointures pour lignes d'adresse de facturation :
-- ligne 1 pour adresse facturation
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AF1
            ON ADF.ADR_ID = AF1.ADR_ID AND AF1.LAD_POSITION = 1
-- ligne 2 pour adresse facturation
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AF2
            ON ADF.ADR_ID = AF2.ADR_ID AND AF2.LAD_POSITION = 2
-- ligne 3 pour adresse facturation
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AF3
            ON ADF.ADR_ID = AF3.ADR_ID AND AF3.LAD_POSITION = 3
-- ligne 4 pour adresse facturation
        LEFT OUTER JOIN T_LIGNE_ADRESSE_LAD AS AF4
            ON ADF.ADR_ID = AF4.ADR_ID AND AF4.LAD_POSITION = 4
GO

/*****
-- CRÉATION DE LA PROCÉDURE STOCKÉE D'INSERTION D'UN OBJET "PERSONNE/ADRESSES"
*****/
-- procédure d'insertion dans les différentes tables
CREATE PROCEDURE P_INSERT_PERSONNE
    @NOM CHAR(32), @PRENOM VARCHAR(25), @TITRE CHAR(16),
    @CP_LIVRAISON CHAR(5), @VILLE_LIVRAISON CHAR(32),
    @LIGNE_ADR_LIVRAISON_1 VARCHAR(32),
    @LIGNE_ADR_LIVRAISON_2 VARCHAR(32),
    @LIGNE_ADR_LIVRAISON_3 VARCHAR(32),
    @LIGNE_ADR_LIVRAISON_4 VARCHAR(32),
    @CP_FACTURATION CHAR(5), @VILLE_FACTURATION CHAR(32),
    @LIGNE_ADR_FACTURATION_1 VARCHAR(32),
    @LIGNE_ADR_FACTURATION_2 VARCHAR(32),
    @LIGNE_ADR_FACTURATION_3 VARCHAR(32),
    @LIGNE_ADR_FACTURATION_4 VARCHAR(32)
AS
BEGIN

DECLARE @TAD_ID INT, @TTR_ID INT, @PRS_ID INT, @ADR_ID INT

```

```
BEGIN TRANSACTION

-- récupération de la clef de titre :
SELECT @TTR_ID = TTR_ID
FROM   T_TITRE_TTR
WHERE  TTR_LIBELLE = @TITRE COLLATE French_CI_AI

-- insertion de la personne :
INSERT INTO T_PERSONNE_PRS (PRS_NOM, PRS_PRENOM, TTR_ID) VALUES (@NOM, @PRENOM,
@TTR_ID)

-- récupération id de personne
SET @PRS_ID = SCOPE_IDENTITY()

-- y a t-il une adresse livraison à insérer ?
IF @CP_LIVRAISON IS NOT NULL AND @VILLE_LIVRAISON IS NOT NULL
BEGIN
-- récupération id de type adresse livraison :
SELECT @TAD_ID = TAD_ID
FROM   T_TYPE_ADRESSE_TAD
WHERE  TAD_LIBELLE = 'Livraison' COLLATE French_CI_AI
-- insertion adresse livraison de cette personne
INSERT INTO T_ADRESSE_ADR (PRS_ID, TAD_ID, ADR_CP, ADR_VILLE)
VALUES (@PRS_ID, @TAD_ID, @CP_LIVRAISON, @VILLE_LIVRAISON)
-- récupération id d'adresse livraison
SET @ADR_ID = SCOPE_IDENTITY()
-- insertions lignes à ligne pour cette adresse de livraison
IF @LIGNE_ADR_LIVRAISON_1 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 1, @LIGNE_ADR_LIVRAISON_1)
IF @LIGNE_ADR_LIVRAISON_2 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 2, @LIGNE_ADR_LIVRAISON_2)
IF @LIGNE_ADR_LIVRAISON_3 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 3, @LIGNE_ADR_LIVRAISON_3)
IF @LIGNE_ADR_LIVRAISON_4 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 4, @LIGNE_ADR_LIVRAISON_4)
END

-- y a t-il une adresse facturation à insérer ?
IF @CP_FACTURATION IS NOT NULL AND @VILLE_FACTURATION IS NOT NULL
BEGIN
-- récupération id de type adresse livraison :
SELECT @TAD_ID = TAD_ID
FROM   T_TYPE_ADRESSE_TAD
WHERE  TAD_LIBELLE = 'facturation' COLLATE French_CI_AI
-- insertion adresse livraison de cette personne
INSERT INTO T_ADRESSE_ADR (PRS_ID, TAD_ID, ADR_CP, ADR_VILLE)
VALUES (@PRS_ID, @TAD_ID, @CP_FACTURATION, @VILLE_FACTURATION)
-- récupération id d'adresse livraison
SET @ADR_ID = SCOPE_IDENTITY()
-- insertions lignes à ligne pour cette adresse de livraison
IF @LIGNE_ADR_FACTURATION_1 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 1, @LIGNE_ADR_FACTURATION_1)
IF @LIGNE_ADR_FACTURATION_2 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 2, @LIGNE_ADR_FACTURATION_2)
IF @LIGNE_ADR_FACTURATION_3 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 3, @LIGNE_ADR_FACTURATION_3)
IF @LIGNE_ADR_FACTURATION_4 IS NOT NULL
INSERT INTO T_LIGNE_ADRESSE_LAD (ADR_ID, LAD_POSITION, LAD_LIGNE)
VALUES (@ADR_ID, 4, @LIGNE_ADR_FACTURATION_4)
END
```

```
COMMIT TRANSACTION

END
GO

/*****
-- CREATION DU TRIGGER REROUTANT L'INSERTION
*****/
-- trigger simulant l'insertion dans la vue
CREATE TRIGGER E_IO_V_PAD
ON V_PERSONNE_ADRESSE_PAD
INSTEAD OF INSERT
AS
BEGIN

DECLARE @NOM CHAR(32), @PRENOM VARCHAR(25), @TITRE CHAR(16),
        @CP_LIVRAISON CHAR(5), @VILLE_LIVRAISON CHAR(32),
        @LIGNE_ADR_LIVRAISON_1 VARCHAR(32),
        @LIGNE_ADR_LIVRAISON_2 VARCHAR(32),
        @LIGNE_ADR_LIVRAISON_3 VARCHAR(32),
        @LIGNE_ADR_LIVRAISON_4 VARCHAR(32),
        @CP_FACTURATION CHAR(5), @VILLE_FACTURATION CHAR(32),
        @LIGNE_ADR_FACTURATION_1 VARCHAR(32),
        @LIGNE_ADR_FACTURATION_2 VARCHAR(32),
        @LIGNE_ADR_FACTURATION_3 VARCHAR(32),
        @LIGNE_ADR_FACTURATION_4 VARCHAR(32);

-- curseur de navigation dans les lignes de la pseudo table d'insertion
DECLARE C CURSOR
FOR
    SELECT PRS_NOM, PRS_PRENOM, TTR_LIBELLE,
           ADR_CP_LIVRAISON, ADR_VILLE_LIVRAISON,
           LAD_LIGNE_LIVRAISON_1, LAD_LIGNE_LIVRAISON_2,
           LAD_LIGNE_LIVRAISON_3, LAD_LIGNE_LIVRAISON_4,
           ADR_CP_FACTURATION, ADR_VILLE_FACTURATION,
           LAD_LIGNE_FACTURATION_1, LAD_LIGNE_FACTURATION_2,
           LAD_LIGNE_FACTURATION_3, LAD_LIGNE_FACTURATION_4
    FROM   INSERTED;

OPEN C

-- récupération des informations d'une ligne
FETCH C INTO @NOM, @PRENOM, @TITRE,
            @CP_LIVRAISON, @VILLE_LIVRAISON,
            @LIGNE_ADR_LIVRAISON_1,
            @LIGNE_ADR_LIVRAISON_2,
            @LIGNE_ADR_LIVRAISON_3,
            @LIGNE_ADR_LIVRAISON_4,
            @CP_FACTURATION, @VILLE_FACTURATION,
            @LIGNE_ADR_FACTURATION_1,
            @LIGNE_ADR_FACTURATION_2,
            @LIGNE_ADR_FACTURATION_3,
            @LIGNE_ADR_FACTURATION_4;

WHILE @@FETCH_STATUS = 0
BEGIN
-- lancement de la procédure d'insertion avec les données d'une ligne
-- de la pseudo table d'insertion dans la vue
EXEC P_INSERT_PERSONNE @NOM, @PRENOM, @TITRE,
                    @CP_LIVRAISON, @VILLE_LIVRAISON,
                    @LIGNE_ADR_LIVRAISON_1,
                    @LIGNE_ADR_LIVRAISON_2,
                    @LIGNE_ADR_LIVRAISON_3,
                    @LIGNE_ADR_LIVRAISON_4,
                    @CP_FACTURATION, @VILLE_FACTURATION,
                    @LIGNE_ADR_FACTURATION_1,
                    @LIGNE_ADR_FACTURATION_2,
                    @LIGNE_ADR_FACTURATION_3,
                    @LIGNE_ADR_FACTURATION_4;
END
```

```

-- récupération des informations d'une ligne
  FETCH C INTO @NOM, @PRENOM, @TITRE,
              @CP_LIVRAISON, @VILLE_LIVRAISON,
              @LIGNE_ADR_LIVRAISON_1,
              @LIGNE_ADR_LIVRAISON_2,
              @LIGNE_ADR_LIVRAISON_3,
              @LIGNE_ADR_LIVRAISON_4,
              @CP_FACTURATION, @VILLE_FACTURATION,
              @LIGNE_ADR_FACTURATION_1,
              @LIGNE_ADR_FACTURATION_2,
              @LIGNE_ADR_FACTURATION_3,
              @LIGNE_ADR_FACTURATION_4;

END

-- fermeture curseur
CLOSE C;

DEALLOCATE C;

END;

/*****
-- TEST D'INSERTION DANS LA VUE MULTITABLE
*****/
-- test d'insertion dans la vue :
INSERT INTO V_PERSONNE_ADRESSE_PAD
  (PRS_ID, PRS_NOM, PRS_PRENOM, TTR_LIBELLE,
   ADR_CP_LIVRAISON, ADR_VILLE_LIVRAISON,
   LAD_LIGNE_LIVRAISON_1, LAD_LIGNE_LIVRAISON_2,
   LAD_LIGNE_LIVRAISON_3, LAD_LIGNE_LIVRAISON_4,
   ADR_CP_FACTURATION, ADR_VILLE_FACTURATION,
   LAD_LIGNE_FACTURATION_1, LAD_LIGNE_FACTURATION_2,
   LAD_LIGNE_FACTURATION_3, LAD_LIGNE_FACTURATION_4)
SELECT 0, 'Dupont', 'Jean', 'monsieur',
       '06000', 'NICE', '65 rue Garnier',
       NULL, NULL, NULL,
       '06000', 'NICE', 'Palais Mare Nostrum',
       '5 promenade des anglais', NULL, NULL

UNION
SELECT 0, 'Martin', 'Martine', 'MADAME',
       '75001', 'PARIS', '18 rue du Louvre',
       NULL, NULL, NULL,
       NULL, NULL, NULL,
       NULL, NULL, NULL

-- résultat
SELECT *
FROM   V_PERSONNE_ADRESSE_PAD

```

Associée à une procédure stockée, le trigger `INSTEAD OF INSERT` reroute l'ensemble des informations à insérer dans les différentes tables depuis la vue, faisant croire à l'utilisateur qu'il travaille une table ordinaire !

Certains outils de mapping relationnel objet permettent d'automatiser cette conception des mises à jours de table...

1.2 - Règles d'intégrité

Elles induisent le comportement des données elles-mêmes (absence de valeur, obligation de valeur, validation de données, unicité, clef...). De ce fait elles doivent être définies et manipulées au niveau du SGBDR et non pas par les applications clientes.

Règles concernées : 3 et 10.

RÈGLE 3 - Gestion du marqueur NULL :

Les marqueurs NULL sont systématiquement pris en charge pour représenter des informations manquantes ou inapplicables, indépendamment du type, du domaine de valeur ou d'une valeur par défaut.

Les modèles précédents avaient le défaut de rendre "obligatoire" la saisie de données. En cas d'impossibilité, une valeur arbitraire (0, chaîne vide, date bidon...) servait de repaire pour indiquer le vide et présentait le défaut de corrompre certains calculs.

DISCUSSION : un tel marqueur est indispensable. En effet comment spécifier l'absence de valeur ? Par une valeur particulière ? Oui, mais si cette valeur particulière apparaît, alors comment distinguer la vraie absence d'information de la fausse ? C'est pourquoi on a ajouté le marqueur NULL. On doit bien dire marqueur et non valeur car justement le NULL représente l'absence de valeur.

De là est née une première difficulté : comment le marqueur NULL doit-il se comporter dans des calculs ? Le principe est simple : le NULL se propage dans les calculs. On dit qu'il est absorbant. Oui, mais que vaut une expression logique contenant un NULL ? Elle vaut ni TRUE, ni FALSE, mais UNKNOWN. Là commence l'univers de la logique trivaluée propre à SQL par exemple...

De plus certains chercheurs ont affirmé que le NULL ne suffisait pas. En effet, le NULL indique qu'il n'y a pas de valeur, mais il n'indique pas *pourquoi* ! Or ce renseignement peut être très important pour le système d'information. Par exemple connaître la couleur du toit d'un véhicule peut s'avérer difficile pour une décapotable lorsque sa capote est abaissée. On pourrait alors mettre un marqueur INCONNU. De même pour une moto, la couleur du toit n'a aucun sens. Il faudrait alors mettre un marqueur INAPPLICABLE. On pourrait rajouter les marqueurs *infinis* : comment spécifier que telle ou telle information s'applique dans le futur ? Par exemple en inventant un marqueur INFINI+...

Hélas ou heureusement, nous ne disposons que du NULL. Hélas car il aurait été intéressant d'en savoir plus sur la cause de l'absence de valeur. Heureusement, car comment seraient évaluées les expressions contenant de tels nouveaux marqueurs ?

Pour toutes informations supplémentaires sur le NULL, voir le papier que j'ai écrit sur SQLpro : <http://sqlpro.developpez.com/cours/null/>

RÈGLE 10 - Indépendance vis-à-vis de l'intégrité :

Les contraintes d'intégrité spécifiques à une base de données relationnelle sont définies à l'aide du langage relationnel et leur définition doit être stockée dans le catalogue et non dans des programmes d'application.

Les contraintes doivent figurer au sein de la base et n'être définies que par le langage relationnel. Elles doivent en outre être décrites dans le catalogue, c'est à dire les tables ou les vues internes (méta données ou tables systèmes) permettant de connaître la structure de la base de données (voir règle 4).

DISCUSSION : cette règle est intéressante à plus d'un titre. Elle impose que toutes les contraintes (domaine, table, assertions) doivent impérativement figurer dans la base de données et non pas dans les programmes applicatifs. Elle est malheureusement souvent violée

par ignorance par des développeurs peu scrupuleux. Et l'oubli d'implantation de ces règles provoque généralement de grands dégâts : piètre qualité des données, doublons sémantiques, références manquantes... Qui finissent par coûter très chers aux entreprises lorsque la base de données est suffisamment volumineuse. Ainsi les études menées sur l'implantation de solutions décisionnelles montrent qu'en moyenne 50% du coût de la solution est dédié au nettoyage des données qui polluent la base. Et c'est justement pour éviter de polluer une base qu'existent les contraintes !

De plus l'intérêt d'avoir la description des contraintes directement accessible dans les méta données de la base, c'est de permettre de faire de la prévention : ainsi en interrogeant les tables système de manière adéquate on peut découvrir quelles sont toutes les contraintes d'une table et préparer côté client des interfaces intelligentes qui aident l'utilisateur à saisir correctement les informations (la présence de l'étoile accolée aux champs de saisie des formulaires web par exemple...) ou encore rectifient à la volée les informations à insérer dans les tables (mise en majuscule, formatage de date, de numéro de téléphone, vérification de la structure d'une adresse mail...).

1.3 - Règles de manipulation des données

Selon Codd, un SGBDR idéal doit prendre en charge de nombreuses fonctionnalités de manipulation de données. L'adhésion à ces règles par un langage de manipulation constitue la complétude du langage.

Règles concernées : 2, 4, 5 et 7.

RÈGLE 2 - Garantie d'accès :

Toute donnée atomique d'une base de données relationnelle est accessible par le biais d'une combinaison du nom de la table ou de la vue, d'une valeur de la clé primaire et d'un nom de colonne.

L'accès à la valeur de toute information atomique est garanti à partir des trois composantes suivante : une table (ou vue) clairement identifié, les valeurs des données d'une clef et le nom d'une colonne.

DISCUSSION : la connaissance de trois informations : nom de la table, valeur(s) de la clef et nom d'une colonne permet de retrouver n'importe quelle information dans une base de données.

Malheureusement on voit trop souvent des bases de données avec des tables sans clef...

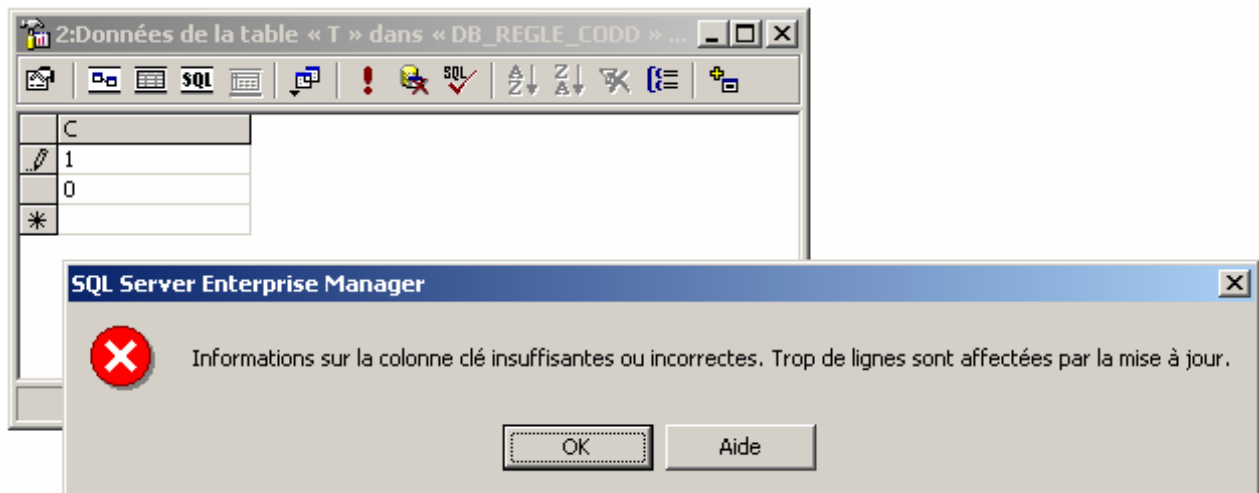
L'absence de clef dans une table pose la plupart du temps des problèmes majeurs voir insolubles.

Exemple :

```
-- créons une table on ne peut plus simple mais sans clef...
CREATE TABLE T (C INT)

-- inserons-y deux lignes identiques...
INSERT INTO T VALUES (0)
INSERT INTO T VALUES (0)
```

Voyons maintenant ce qui se passe dans une interface graphique lorsque je veut mettre à jour l'une des deux lignes :



Nous savons que par essence les bases de données sont ensemblistes puisque l'algèbre relationnel est un sous ensemble pur et dur des ensembles mathématiques. Or les IHM passent généralement par des curseurs SQL, et tentent d'effectuer une mise à jour pour la ligne courante (CURRENT OF). Mais comme l'ordre de mise à jour reste l'UPDATE et que le seul moyen de faire un UPDATE sur une ligne précise consiste à fournir dans la clause WHERE suffisamment d'information pour ne toucher que les lignes impactées, alors SQL se trouve devant une contradiction : mettre à jour une seule ligne ou plusieurs ? Beaucoup de développeurs débutants pensent naïvement qu'il existe une quelconque numérotation interne des lignes. Un tel système irait à l'encontre des principes mêmes des SGBDR dans lesquels toute notion d'ordre implicite est par nature bannie, mais aussi grèverait les performances du système !

RÈGLE 4 - Catalogue relationnel, dynamique et accessible directement :

La description de la base de données et de son contenu est représentée au niveau logique de la même manière que les données ordinaires (des tables).

Ce qui signifie qu'il existe un langage uniforme pour la manipulation des données comme pour l'accès aux méta données (la description de la structure et des objets de la base) et une seule et même façon logique de présenter ces informations : par des tables ou des vues.

DISCUSSION : simple, logique, direct et efficace.... pour décrire le contenu d'une base de données, rien ne vaut la base de données elle même ! La norme SQL a même entériné la forme avec laquelle la chose doit s'exprimer : les fameuses vues d'information de schéma...

En voici les principales :

```
-- pour les domaines :
INFORMATION_SCHEMA.DOMAINS
INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS

-- pour les schémas :
INFORMATION_SCHEMA.SCHEMATA

-- pour les tables :
INFORMATION_SCHEMA.TABLES
INFORMATION_SCHEMA.TABLE_PRIVILEGES
INFORMATION_SCHEMA.TABLE_CONSTRAINTS
```

```
-- pour les contraintes de validation :  
INFORMATION_SCHEMA.CHECK_CONSTRAINTS  
  
-- pour les contraintes de table :  
INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE  
INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE  
INFORMATION_SCHEMA.KEY_COLUMN_USAGE  
INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS  
  
-- pour les vues :  
INFORMATION_SCHEMA.VIEWS  
INFORMATION_SCHEMA.VIEW_COLUMN_USAGE  
INFORMATION_SCHEMA.VIEW_TABLE_USAGE  
  
-- pour les colonnes des tables ou des vues :  
INFORMATION_SCHEMA.COLUMNS  
INFORMATION_SCHEMA.COLUMN_PRIVILEGES  
INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE  
  
-- pour les procédures et fonctions :  
INFORMATION_SCHEMA.ROUTINES  
INFORMATION_SCHEMA.ROUTINE_COLUMNS  
INFORMATION_SCHEMA.PARAMETERS
```

Et dire qu'il y a encore des SGBDR qui n'en font qu'à leur tête en proposant des commandes spécifiques, incompatible avec les autres. Par exemple le fameux "SHOW TABLE" de MySQL... Encore peut-on comprendre la réalisation de certaines procédures stockées en sus des vues normalisées. Mais de là à mettre des pseudo commande SQL, cela frise, à mon sens, l'escroquerie ! C'est d'ailleurs toujours la même histoire depuis les débuts de l'informatique : rendre les choses incompatible avec les autres, même au mépris des normes, et imposer son "standard" afin de rendre le plus difficile possible la portabilité... IBM nous à déjà servit cette soupe et perdu la partie. Oracle l'a longtemps fait avant, aujourd'hui, de s'en mordre les doigts. Microsoft l'a bien tenté, mais s'est repris juste à temps...

RÈGLE 5 - Langage de manipulation de données complet :

Au moins un des langages du SGBDR doit avoir une syntaxe complète et doit permettre (1) la définition des données, (2) la formation des vues, (3) la manipulation des données; (4) la gestion des règles d'intégrité, (5) les autorisations et (6) les frontières des transactions.

Le langage normatif SQL assure toutes ces fonctions à travers différentes facettes :

- DDL (Data Definition Langage : CREATE, ALTER, DROP) : 1, 2, 4;
- DML (Data Manipulation Langage : SELECT, INSERT, DELETE, UPDATE) : 3;
- DCL (Data Control Langage : GRANT, REVOKE) : 5;
- TCL (Transaction Control Langage : BEGIN TRANSACTION, COMMIT, ROLLBACK) : 6.

Mais des éléments DML sont nécessaires pour définir certaines contraintes SQL.

Tout langage de pilotage d'une base de données relationnelle doit comporter au moins ces éléments là. J'ai longtemps dit que mySQL n'était pas pour moi un SGBD relationnel. Je l'ai dit tout le temps ou MySQL a été incapable d'assurer la gestion des transactions. Ce n'est heureusement plus le cas et mySQL vient, avec plus de 20 ans de retard, de rejoindre le club des éditeurs de SGBDR. Cependant il existe toujours des produits qui se disent SGBD relationnel et n'offrent pas le support des transactions (Access, SQL Lite par exemple).

RÈGLE 7 - Insertion, suppression et modification ensemblistes :

Le SGBDR retourne un ensemble d'éléments en réponse aux requêtes qui lui sont soumises. Il doit pouvoir mettre à jour un ensemble d'éléments en exécutant une seule requête.

Cette règle renforce la nature ensembliste d'un SGBDR. La manipulation des données doit se faire aussi bien sur les tables de base que sur les vues, dans la limite de la règle 6.

DISCUSSION : regardons ce qui se passe de différent entre un langage, qui, par nature, opère de manière ensembliste (comme SQL) et un langage itératif...

Exemple (similaire à celui de la règle 12) :

```
-- créons une table dotée d'une unique colonne
-- les valeurs de cette colonne étant impérativement unique...
CREATE TABLE T_SET (VAL INT UNIQUE)
-- insérons quelques valeurs qui respectent l'unicité :
INSERT INTO T_SET VALUES (-8)
INSERT INTO T_SET VALUES (-7)
INSERT INTO T_SET VALUES (-6)
INSERT INTO T_SET VALUES (-5)
INSERT INTO T_SET VALUES (-4)
INSERT INTO T_SET VALUES (-3)
INSERT INTO T_SET VALUES (-2)
INSERT INTO T_SET VALUES (-1)
INSERT INTO T_SET VALUES (0)
INSERT INTO T_SET VALUES (1)
INSERT INTO T_SET VALUES (2)
INSERT INTO T_SET VALUES (3)
INSERT INTO T_SET VALUES (4)
INSERT INTO T_SET VALUES (5)
INSERT INTO T_SET VALUES (6)
INSERT INTO T_SET VALUES (7)
INSERT INTO T_SET VALUES (8)
```

Le problème consiste maintenant à mettre à jour la colonne VAL avec l'opération $VAL = VAL^3$.

Par itération, du fait de la contrainte UNIQUE, il n'est pas possible de réaliser un tel calcul. En effet si l'on réalise une boucle montante, 2 au cube = 8 et il y a violation de la contrainte d'unicité. De la même manière avec une boucle descendante, -2 au cube = -8 et de nouveau la contrainte d'unicité est violée !

En revanche, un tel UPDATE SQL réussit parfaitement du fait de la nature ensembliste de la mise à jour. Comprenez donc que la validation de la contrainte se fait sur l'ensemble de données mis à jour et non pas à chaque ligne...

```
UPDATE T_SET
SET VAL = POWER(VAL, 3)
```

1.4 - Règles d'indépendance des données

Ces règles garantissent que pour tout utilisateur comme pour tout développeur, aucune application n'a besoin d'être réécrite si des réorganisations de la base de données sont effectuées.

Règles concernées : 8, 9 et 11.

RÈGLE 8 - Indépendance physique des données :

Les applications et les programmes terminaux sont logiquement inaffectés lorsque les méthodes d'accès physiques ou les structures de stockage sont modifiées.

La modification des emplacements de stockage des données (transferts, renommage, exportation, etc.), ne doivent en aucun cas rejaillir sur l'écriture des programmes clients. Ils ne doivent pas non plus entacher le code existant.

DISCUSSION : contrairement à une application qui ouvrirait un fichier pour lire et écrire des données, les chemins physiques des fichiers contenant les données de la base, comme la nature des supports, ne doivent en aucun cas être pris en compte pour la manipulation des données. Autrefois, l'usage de langages gérant des fichiers, comme COBOL, nécessitait d'écrire en dur dans le programme le chemin d'accès au fichier. Cela a perduré jusqu'au début des bases de données *réseau*.

RÈGLE 9 - Indépendance logique des données :

Les applications et les programmes terminaux sont logiquement inaffectés, quand des changements de tous ordres, préservant les informations et qui ne leur portent théoriquement aucune atteinte, sont apportés aux tables de base (restructuration).

Une modification de la structure de la table, sur des éléments qui n'ont pas été utilisés dans le code, ne doit pas perturber le fonctionnement de l'existant.

DISCUSSION : ainsi, il doit ainsi être possible de rajouter une colonne à une table, de renommer une colonne, d'ajouter une table ou une vue à une base, etc, sans perturber le fonctionnement des applications. Mieux, si une colonne d'une table n'est pas utilisée, alors il est possible de la supprimer sans affecter les programmes terminaux.

Je suis souvent stupéfait de voir que de nombreux développeurs qui utilisent une base de données relationnelle et à qui l'on conseille de rajouter telle ou telle colonne, vue ou table dans leur base, s'y opposent formellement. Il n'y a aucune logique là dedans. En quoi l'ajout d'un objet qui n'est pas déjà référencé par une application aurait-il une influence jusqu'à abîmer les données de la base ? On nage en plein obscurantisme !

En revanche, le langage SQL pose des problèmes qui peuvent faire que cette règle soit difficile à appliquer. Là encore il s'agit d'un problème de méconnaissance du développement avec un SGBDR.

Regardons par exemple, les constructions de requêtes suivantes :

```
SELECT * FROM...
INSERT INTO <table> VALUES (... )
ORDER BY 1, 2, 3, ... n
```

Elles peuvent avoir des effets indésirables lors de la modification de la structure d'une base de données. C'est pourquoi à l'exception de la mise au point du code, ces formes d'expressions SQL ne devraient jamais être utilisés en production. En particulier le SELECT * devrait être interdit dans l'écriture du code des applications.

RÈGLE 11 - Indépendance de distribution :

Le langage relationnel doit permettre aux programmes d'application et aux requêtes de demeurer identiques sur le plan logique lorsque des données, quelles qu'elles soient, sont physiquement réparties ou centralisées.

Les opérations de répartition de données (partition de tables, de bases...), comme les opérations de regroupement des données (concaténation, agrégation...) ne doivent pas entraîner de dysfonctionnement des applications clientes.

DISCUSSION : là nous entrons déjà dans le domaine de l'administration des bases de données. Comment faire lorsqu'une base de données dépasse la capacité des plus gros disques disponible ? Peut-on répartir les données d'une base sur différents disques ? Peut-on même imposer que telle ou telle table soit explicitement placée sur tel ou tel fichier de tel ou tel disque ?

Tout cela est aujourd'hui possible, et ces manoeuvres peuvent même être exécutées à chaud sur les meilleurs serveurs.

Exemple (spécifique à MS SQL Server 2005) :

```
-- création d'une base multi fichier, multi disques
CREATE DATABASE DB_TEST_FILES
ON
PRIMARY
(NAME = DATAFILE0,
 FILENAME = 'C:\SQL_DATABASES\DATAFILE0.mdf',
 SIZE = 10MB),
FILEGROUP FG_DATA1
(NAME = DATAFILE1,
 FILENAME = 'D:\SQL_DATABASES\DATAFILE1.mdf',
 SIZE = 100MB),
FILEGROUP FG_DATA2
(NAME = DATAFILE2,
 FILENAME = 'E:\SQL_DATABASES\DATAFILE2.mdf',
 SIZE = 100MB),
FILEGROUP FG_DATA3
(NAME = DATAFILE3,
 FILENAME = 'F:\SQL_DATABASES\DATAFILE3.mdf',
 SIZE = 100MB),
FILEGROUP FG_DATA4
(NAME = DATAFILE4,
 FILENAME = 'G:\SQL_DATABASES\DATAFILE4.mdf',
 SIZE = 100MB)
GO
-- si vous ne disposez pas des disques D, E, F et G placez tout sur le C.
-- n'oubliez pas de créer le répertoire SQL_DATABASES à la racine de chaque disque

USE DB_TEST_FILES
GO

-- création d'une fonction de repartition des données
CREATE PARTITION FUNCTION PF_AN (DATETIME)
AS RANGE RIGHT
FOR VALUES ('20060101', '20070101')
GO
-- pour 2 valeurs pivot (borne droite incluse) il nous faut 3 espaces de stockage

-- création d'un schéma de répartition des données reprenant la fonction de
répartition
CREATE PARTITION SCHEME PS_AN
AS PARTITION PF_AN
TO (FG_DATA1, FG_DATA2, FG_DATA3, FG_DATA4)
GO
Le schéma de partition 'PS_AN' a été créé avec succès.
'FG_DATA4' est marqué comme étant le prochain groupe de fichiers utilisé dans le
schéma de partition 'PS_AN'.
-- message normal : nous avons créé 4 espaces de stockage. L'un servira de réserve
```

```

-- création d'une table sur le partitionnement préalablement défini.
CREATE TABLE T_FACTURE_FCT
(FCT_ID          INT NOT NULL IDENTITY,
 FCT_DATE       DATETIME NOT NULL,
 FCT_DATA       UNIQUEIDENTIFIER DEFAULT NEWID())
ON PS_AN(FCT_DATE)
-- les données des années avant 2006 iront dans le groupe de fichier DATAFILE1,
-- donc actuellement dans le fichier D:\SQL_DATABASES\DATAFILE1.mdf
-- les données de l'année 2006 iront dans le groupe de fichier DATAFILE2,
-- donc actuellement dans le fichier E:\SQL_DATABASES\DATAFILE2.mdf
-- les données des années après 2006 iront dans le groupe de fichier DATAFILE3,
-- donc actuellement dans le fichier F:\SQL_DATABASES\DATAFILE3.mdf

-- vérifions la répartition des données en créant des lignes
-- pour toutes les dates de 2005 à 2008
DECLARE @D DATETIME
SET @D = '20050101'
WHILE @D < '20090101'
BEGIN
    INSERT INTO T_FACTURE_FCT VALUES (@D, DEFAULT)
    SET @D = @D + 1
END

-- pour corser le tout, rajoutons quelques lignes avec ces mêmes dates :
INSERT INTO T_FACTURE_FCT (FCT_DATE)
SELECT FCT_DATE
FROM   T_FACTURE_FCT
GO 9
-- hé oui, cela peut prendre un peu de temps !

-- vérification de la répartition des données :
SELECT $partition.PF_AN(FCT_DATE) AS PARTITION_NUMBER,
       COUNT(*) As NOMBRE,
       MIN(FCT_DATE) AS DATE_MIN, MAX(FCT_DATE) AS DATE_MAX
FROM   T_FACTURE_FCT
GROUP BY $partition.PF_AN(FCT_DATE)
ORDER BY 1

PARTITION_NUMBER NOMBRE          DATE_MIN          DATE_MAX
-----
1                 186880          2005-01-01 00:00:00.000 2005-12-31 00:00:00.000
2                 186880          2006-01-01 00:00:00.000 2006-12-31 00:00:00.000
3                 374272          2007-01-01 00:00:00.000 2008-12-31 00:00:00.000

-- décidons maintenant de placer les données de 2008 dans la partition de réserve
ALTER PARTITION FUNCTION PF_AN()
SPLIT RANGE ( '20080101');

-- vérification de la nouvelle répartition des données :
SELECT $partition.PF_AN(FCT_DATE) AS PARTITION_NUMBER,
       COUNT(*) As NOMBRE,
       MIN(FCT_DATE) AS DATE_MIN, MAX(FCT_DATE) AS DATE_MAX
FROM   T_FACTURE_FCT
GROUP BY $partition.PF_AN(FCT_DATE)
ORDER BY 1

PARTITION_NUMBER NOMBRE          DATE_MIN          DATE_MAX
-----
1                 186880          2005-01-01 00:00:00.000 2005-12-31 00:00:00.000
2                 186880          2006-01-01 00:00:00.000 2006-12-31 00:00:00.000
3                 186880          2007-01-01 00:00:00.000 2007-12-31 00:00:00.000
4                 187392          2008-01-01 00:00:00.000 2008-12-31 00:00:00.000

-- décidons maintenant de placer les données antérieure à 2007
-- dans la partition basse
ALTER PARTITION FUNCTION PF_AN()
MERGE RANGE ( '20060101');

-- vérification de la nouvelle répartition des données :

```

```

SELECT $partition.PF_AN(FCT_DATE) AS PARTITION_NUMBER,
       COUNT(*) As NOMBRE,
       MIN(FCT_DATE) AS DATE_MIN, MAX(FCT_DATE) AS DATE_MAX
FROM   T_FACTURE_FCT
GROUP BY $partition.PF_AN(FCT_DATE)
ORDER BY 1

```

PARTITION_NUMBER	NOMBRE	DATE_MIN	DATE_MAX
1	373760	2005-01-01 00:00:00.000	2006-12-31 00:00:00.000
2	186880	2007-01-01 00:00:00.000	2007-12-31 00:00:00.000
3	187392	2008-01-01 00:00:00.000	2008-12-31 00:00:00.000

```
-- nous venons de faire glisser des données d'une partition à l'autre.
```

Ainsi on peut imaginer une comptabilité dont les exercices clos sont stockés dans une partition particulière voire en read only, et l'exercice actif dans une autre. Comme la clôture s'effectue toujours avec quelques mois de retard, il nous faut 3 partitions :

- 1) les exercices clos
- 2) l'exercice en cours de clôture
- 3) l'exercice actif

Enfin pour assurer ce glissement il faut une partition supplémentaire d'échange.

C'est exactement ce que nous venons de voir !

Et pendant le temps que les glissements de partitions s'effectuent, les utilisateurs continuent à travailler comme si de rien n'était et les développeurs peuvent continuer à coder sans se soucier de là où se trouvent les données. Tout cela se faisant bien entendu à chaud et en tâche de fond, c'est à dire en minimisant les ressources afin que le service des données apparaisse avec des temps de réponse ordinaires...

Au fait où sont situées les données des partitions ?

```

SELECT PF.name AS PARTITION_NAME, PS.NAME AS PARTITION_SCHEMA,
       DS.name AS FILE_GROUP, T.PARTITION_NUMBER,
       T.NOMBRE, T.DATE_MIN, T.DATE_MAX,
       physical_name AS PHYSICAL_FILE
FROM   sys.partition_functions AS PF
       INNER JOIN sys.partition_schemes AS PS
           ON PF.function_id = PS.function_id
       INNER JOIN sys.destination_data_spaces AS DDS
           ON PS.data_space_id = DDS.partition_scheme_id
       INNER JOIN sys.data_spaces AS DS
           ON DDS.data_space_id = DS.data_space_id
       INNER JOIN sys.database_files AS DF
           ON ds.data_space_id = DF.data_space_id
       INNER JOIN (SELECT $partition.PF_AN(FCT_DATE) AS PARTITION_NUMBER,
                           COUNT(*) As NOMBRE,
                           MIN(FCT_DATE) AS DATE_MIN, MAX(FCT_DATE) AS DATE_MAX
                   FROM   T_FACTURE_FCT AS F
                   GROUP BY $partition.PF_AN(FCT_DATE)) AS T
       ON T.PARTITION_NUMBER = DDS.destination_id
WHERE  PF.name = 'PF_AN'

```


2 - PLUS ENCORE...

En 1990, Codd étendit ses 12 règles à 18 en rajoutant des éléments sur les concepts de catalogue, domaines, privilèges etc...

Hugh Date et Chris Darwen, critiquèrent les bases de données purement relationnelles dans « The Third Manifesto », au profit de bases de données intégrant des objets plus larges, violant ainsi la notion d'atomicité des données, mais ouvrant la voie au concept du « relationnel objet ». Tant est si bien qu'aujourd'hui on peut considérer la notion de SGBD *réellement relationnel* (qui respectent les règles en n'en enfreignant aucune) et SGBD *pseudo relationnel* (qui respecte la plupart de ces règles, mais en enfreint certaines).

Enfin en 1992, Codd édicta 12 nouvelles règles concernant les bases de données multidimensionnelles reposant sur le modèle OLAP.

Il est cependant certain qu'un bon SGBD relationnel doit s'éloigner le moins possible de ces concepts et conserver l'aspect ensembliste comme règle absolue. D'ailleurs, chaque fois que l'on s'éloigne de ces principes, il y a de grandes chances que les performances de l'outil comme la cohérence des données s'en ressentent !



SQLspot : un focus sur vos données !

SQLSPOT vous apporte les solutions dont vous avez besoin pour vos bases de données **Microsoft SQL Server**

GAGNEZ DU TEMPS ET DE L'ARGENT

pour toutes vos problématiques Microsoft SQL server avec **Frédéric BROUARD**, expert SQL Server, enseignant aux Arts & Métiers et à l'Institut Supérieur d'Électronique et du Numérique (Toulon).

Tél. : **06 11 86 40 66**


Interventions sur Nice, Aix, Marseille, Toulouse, Lyon, Nantes, Paris...

SQLspot a été créée en mars 2007 à l'initiative de Frédéric Brouard, après trois ans d'activité sur le conseil en matière de SGBDR SQL Server, afin de proposer des services à valeur ajoutée à la problématique des données de l'entreprise :

- conseil (par exemple stratégie de gestion des données),
- modélisation de données (modèles conceptuels, logiques et physiques, rétro ingénierie...),
- qualification des données (validation, vérifications, reformatage automatique de données...),
- réalisation d'algorithmes de traitement de données (indexation textuelle avancée, gestion de méta modèles, traitements récursif de données arborescentes ou en graphe...),
- formation (aux concepts des SGBDR, au langage SQL, à la modélisation de données, à SQL Server ...)
- audit (audit de structure de base de données, de serveur de données, d'architecture de données...)
- tuning (affinage des paramètres OS, réseau et serveur pour une exploitation au mieux des ressources)
- optimisation (réécriture de requêtes, étude d'indexation, maintenance de données, refonte de code serveur...)

Vos données constituent le capital essentiel de votre système informatique. Pensez à les entretenir aussi bien que le reste...

mail : SQLpro@SQLspot.com



<http://www.sqlspot.com>