De l'intérêt des règles d'intégrité dans un SGBD

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 17 janvier 2008

https://www.bortzmeyer.org/regles-integrite-sgbd.html

Il semble que l'utilisation des règles d'intégrité dans les SGBD se fasse plus rare depuis quelques années. C'est dommage, car ces règles permettent un accès bien plus sécurisé à la base de données et documentent le schéma. Pourquoi mettre de telles règles et comment ?

Tous les SGBD (même MySQL, longtemps en retard, s'y est mis) permettent d'exprimer des **contraintes d'intégrité**, permettant d'exprimer les règles que les données de la base doivent respecter. Ces règles permettent d'augmenter la sécurité en limitant le risque que des commandes SQL boguées ne laissent la base dans un état incorrect. SQL offre plusieurs moyens (pas tous standards) pour décrire ces contraintes.

Si on utilise un SGBD, c'est probablement parce les données qu'on lui confie sont importantes. On ne souhaite donc pas qu'elles soient modifiées librement, elles doivent, à tout moment, respecter un certain nombre de **règles**, par exemple que le code postal soit présent dans une liste des codes postaux effectivement alloués, ou, au minimum, qu'il soit composé de cinq chiffres.

Il y a plusieurs endroits pour mettre de telles règles. Si un seul programme accède à la base en écriture, on peut mettre ces règles dans le programme (qu'il soit en Cobol, en Java ou en Perl). Mais c'est une contrainte ennuyeuse : si on veut développer un second programme, on doit remettre toutes les contraintes. Plus récemment, l'utilisation massive de "*middleware*" d'accès à la base de données a poussé certains développeurs à mettre toutes les règles d'intégrité dans ledit "*middleware*" : même si plusieurs programmes accèdent à la base, tous passent par le "*middleware*" et tous vont donc devoir respecter les règles.

Ce développement du "*middleware*" et l'influence de logiciels simples n'incluant pas toutes les fonctions d'un SGBD (comme MySQL) a fait reculer l'utilisation de contraintes d'intégrité dans le SGBD lui-même. C'est dommage. En effet, mettre le plus possible de contraintes dans le SGBD, en SQL (avec extensions), permet d'utiliser plusieurs programmes radicalement différents pour accéder à la base, sans crainte pour la cohérence des données, ce qui donne plus de souplesse au système d'information. En outre, cela permet d'utiliser toutes les possibilités de SQL. Si, par exemple, ce qui arrive parfois, on doit modifier un grand nombre des données de la base pour un changement ou un problème imprévu, écrire une requête SQL est plus facile que de modifier le "*middleware*" pour l'adapter à ce nouveau besoin, non

prévu, et qui n'arrivera qu'une seule fois. La grande force de SQL est justement de pouvoir facilement exprimer des requêtes non envisagées à l'origine. Cette possibilité soulève parfois de l'inquiétude, mais les règles d'intégrité permettent de travailler, même en SQL, avec un bon filet de sécurité.

Naturellement, aucun filet de sécurité n'est parfait. Toutes les règles ne peuvent pas être exprimées dans la base (le SQL standard n'est pas un langage de Turing) et, donc, il faudra quand même mettre certaines règles dans un langage classique. Mais je trouve la sécurité des règles mises dans le SGBD très rassurante : d'une certaine façon, la base de données se défend toute seule contre les programmes écrits un peu trop hâtivement.

Comment programme t-on des règles d'intégrité ? Il existe de nombreuses méthodes. Commençons par les plus standard, qui font partie de SQL. Tous les exemples sont faits avec PostgreSQL et les liens pointent vers la documentation de PostgreSQL.

Si une donnée doit être présente une fois et une seule dans une table, le mot-clé `UNIQUE` permet de faire respecter cette règle. Par exemple, un registre de noms de domaine peut avoir une règle analogue, puisque les noms sont uniques.

```
essais=> CREATE TABLE Domaines (nom TEXT UNIQUE);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "domaines_nom_key" for table "domaines"
CREATE TABLE
essais=> INSERT INTO Domaines VALUES ('foobar.example');
INSERT 372690 1
essais=> INSERT INTO Domaines VALUES ('autre.example');
INSERT 372691 1
essais=> INSERT INTO Domaines VALUES ('foobar.example');
ERROR:  duplicate key violates unique constraint "domaines_nom_key"
```

Si une donnée doit appartenir à un ensemble bien défini de valeurs, le système de typage de SQL <http://www.postgresql.org/docs/current/interactive/datatype.html> peut aider. Ainsi, on peut préciser qu'une donnée doit être une date <http://www.postgresql.org/docs/current/interactive/datatype-datetime.html> (la syntaxe d'entrée est ISO 8601) :

```
essais=> CREATE TABLE Nouvelles (pub DATE);
CREATE TABLE
essais=> INSERT INTO Nouvelles VALUES ('2007-1-16');
INSERT 372695 1
essais=> INSERT INTO Nouvelles VALUES ('1879-8-8');
INSERT 372696 1
essais=> INSERT INTO Nouvelles VALUES ('2000-13-32');
ERROR:  date/time field value out of range: "2000-13-32"
```

On est ainsi protégé contre les erreurs de saisie ou de calcul.

PostgreSQL permet de créer ses propres types <http://www.postgresql.org/docs/current/interactive/xtypes.html>, ce qui permet d'obtenir ces contraintes de types (SQL dit de **domaines**, le terme **types** provenant plutôt de la programmation classique) avec des types quelconques. Outre les types standards de SQL comme l'entier ou la chaîne de caractères, PostgreSQL dispose d'une grande

variété de types comme un type pour les adresses IP, IPv4 et IPv6. (Je cite cet exemple car j'ai vu beaucoup d'applications où un champ censé stocker une adresse IP était une simple chaîne de caractères, sans contrôle particulier ou, à peine mieux, un contrôle uniquement dans le client Web via un peu de Javascript, ce qui permet au client Web d'introduire facilement n'importe quelle valeur, juste en coupant Javascript.)

SQL permet également d'imposer qu'une **référence** à un tuple pointe réellement vers un tuple existent (contrainte référentielle).

```
essais=> CREATE TABLE Catalogue (ref VARCHAR(6) UNIQUE, description TEXT);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "catalogue_ref_key" for table "catalogue"
CREATE TABLE
essais=> INSERT INTO Catalogue VALUES ('456223', 'Lampe de chevet');
INSERT 372714 1
essais=> INSERT INTO Catalogue VALUES ('6778', 'Lampe de poche');
INSERT 372715 1
essais=> INSERT INTO Catalogue VALUES ('99115', 'Allumettes');
INSERT 372716 1

essais=> CREATE TABLE Commandes (quand DATE, quoi VARCHAR(6) REFERENCES Catalogue(ref));
CREATE TABLE
essais=> INSERT INTO Commandes VALUES ('2007-1-17', '6778');
INSERT 372723 1
essais=> INSERT INTO Commandes VALUES ('2007-1-17', '1');
ERROR:  insert or update on table "commandes" violates foreign key constraint "$1"
DETAIL:  Key (quoi)=(1) is not present in table "catalogue".
```

On note que ces contraintes ne s'appliquent pas qu'à la création (elles suivent l'esprit SQL, ce sont des déclarations, des assertions sur la base, pas des instructions à exécuter). Elles empêchent également de « scier la branche sur laquelle on est assis », par exemple en détruisant des données qu'on référence encore :

```
essais=> DELETE FROM Catalogue WHERE ref = '6778';
ERROR:  update or delete on "catalogue" violates foreign key constraint "$1" on "commandes"
DETAIL:  Key (ref)=(6778) is still referenced from table "commandes".
```

Si les contrôles à effectuer sont un peu plus compliquées, et qu'on ne souhaite pas forcément créer ses propres types, `CHECK` permet bien des choses :

```
essais=> CREATE TABLE Employes (salaire NUMERIC CHECK (salaire > 1280));
                                                        -- Le SMIC
CREATE TABLE
essais=> INSERT INTO Employes VALUES (3150);
INSERT 372735 1
essais=> INSERT INTO Employes VALUES (1291);
INSERT 372736 1
essais=> INSERT INTO Employes VALUES (940);
ERROR:  new row for relation "employes" violates check constraint "employes_salaire"
```

Malgré les efforts de Parisot, le SGBD refusera des salaires inférieurs au SMIC.

Les tests faits avec `CHECK` peuvent être plus complexes, ici on va tester que l'adresse de courrier électronique est valable (**attention** : beaucoup de logiciels mettent en œuvre ce test n'importe comment <https://www.bortzmeyer.org/arreter-d-interdire-des-adresses-legales.html>.) On note aussi qu'un nom a été donné au test, pour améliorer les messages d'erreur :

```
essais=> CREATE TABLE Contacts (adresse TEXT NOT NULL
essais(>     CONSTRAINT Adresse_valide CHECK (adresse ~ '^.+@.+$'));
CREATE TABLE
essais=> INSERT INTO Contacts VALUES ('stephane+blog@bortzmeyer.org');
INSERT 372743 1
essais=> INSERT INTO Contacts VALUES ('postmaster@hotmail.com');
INSERT 372744 1
essais=> INSERT INTO Contacts VALUES ('ano nymous');
ERROR:  new row for relation "contacts" violates check constraint "adresse_valide"
```

On peut tester de manière analogue, toujours avec des expressions rationnelles, le numéro de téléphone :

```
essais=> CREATE TABLE Contacts (telephone TEXT NOT NULL
essais(>      CONSTRAINT Numero_telephone CHECK (telephone ~ '^\\+[0-9]+[ 0-9]+$'));
CREATE TABLE
essais=> INSERT INTO Contacts VALUES ('+33 1 39 30 83 46');
INSERT 372751 1
essais=> INSERT INTO Contacts VALUES ('+1 123 456');
INSERT 372752 1
essais=> INSERT INTO Contacts VALUES ('22');
ERROR:  new row for relation "contacts" violates check constraint "numero_telephone"
```

Cette syntaxe (à la norme E.164) peut être jugée trop rigide pour les utilisateurs (par exemple elle oblige à entrer le code international, 33 pour la France, systématiquement). On peut autoriser des syntaxes relatives en entrée mais, à mon avis, pas dans la base, qui doit rester le plus homogène possible. Normaliser un numéro relatif comme `01 39 30 83 46`) en numéro E.164 (`+33 1 39 30 83 46`), accepter des points ou des virgules, etc, doit plutôt se faire dans l'interface utilisateur.

Autre utilisation de `CHECK` : SQL ne dispose pas de types énumérés, comme beaucoup de langages de programmation `CHECK` est souvent un moyen utilisé pour les mettre en œuvre :

```
essais=> CREATE TABLE Personne (statut TEXT
essais(>      CHECK (statut IN ('prospect', 'fournisseur', 'client', 'presse')));
CREATE TABLE
essais=> INSERT INTO Personne VALUES ('fournisseur');
INSERT 372759 1
essais=> INSERT INTO Personne VALUES ('client');
INSERT 372760 1
essais=> INSERT INTO Personne VALUES ('autre');
ERROR:  new row for relation "personne" violates check constraint "personne_statut"
```

Il existe des cas où la syntaxe simple de `CHECK` ou des contraintes référentielles ne suffit pas. Par exemple, un registre de noms de domaines veut faire respecter la syntaxe décrite dans les RFC 1034 [1] et RFC 1123. Les expressions rationnelles comme `^[a-z0-9-\.]+$` ne suffisent pas car elles autorisent des noms illégaux comme `pas--terrible..fr`. Il faut alors écrire une fonction et s'assurer qu'elle soit appelée avant la création d'un nom. Même chose si la règle dépend de valeurs stockée dans la base de données, autres que la simple référence à une valeur existante. Par exemple, si on veut refuser l'enregistrement d'une commande si la trésorerie est à moins de 10 000 €, il faudra aussi utiliser une fonction.

Voici un exemple de fonction pour tester la syntaxe que le registre de noms de domaine accepte (elle est souvent plus restrictive que ce que le DNS accepte) :

```
-- Only a small part of the real checks!

CREATE OR REPLACE FUNCTION legal_name(TEXT) RETURNS BOOLEAN
  AS 'DECLARE
            name_to_check TEXT;
            next_dot INTEGER;
            two_dots INTEGER;
      BEGIN
        SELECT INTO name_to_check lower($1);
        IF name_to_check !~ ''^[\.0-9a-zA-Z\-]+$'' THEN
           RETURN FALSE; -- This test could have been done in a simple CHECK
        END IF;
        two_dots := strpos(name_to_check, ''..'');
        IF two_dots != 0 THEN
           RETURN FALSE;
        END IF;
-- Other tests can be inserted here
        RETURN TRUE;
       END;'
     LANGUAGE PLPGSQL;
```

Pour que cette fonction soit appelée avant la création du nom de domaine, on utilise les déclencheurs (*"triggers"*) :

```
CREATE OR REPLACE FUNCTION check_legal() RETURNS TRIGGER
    AS 'BEGIN
          IF NOT legal_name(NEW.name) THEN
            RAISE EXCEPTION ''Illegal syntax for a domain name'';
          END IF;
          RETURN NEW;
        END;'
    LANGUAGE PLPGSQL;

DROP TRIGGER check_legal ON Domains;
CREATE TRIGGER check_legal
  BEFORE INSERT OR UPDATE ON Domains
  FOR EACH ROW
  EXECUTE PROCEDURE check_legal();
```

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc1034.txt`

Ainsi, toute tentative d'insérer (ou de modifier) un nom de domaine va appeler la function `check_legal` :

```
essais=> CREATE TABLE Domains (name TEXT UNIQUE NOT NULL);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "domains_name_key" for table "domains"
CREATE TABLE
essais=> \i mychecks.sql
CREATE FUNCTION
CREATE FUNCTION
CREATE TRIGGER
essais=> INSERT INTO Domains VALUES ('foobar.example');
INSERT 372800 1
essais=> INSERT INTO Domains VALUES ('foobar..example');
ERROR:  Illegal syntax for a domain name
essais=> INSERT INTO Domains VALUES ('tyuyty ghg');
ERROR:  Illegal syntax for a domain name
```

Pour le cas où on veut utiliser des données qui se trouvent dans la base, on définit fonction et déclencheur :

```
CREATE OR REPLACE FUNCTION enough_money(INTEGER) RETURNS BOOLEAN
  AS 'DECLARE
        amount_left INTEGER;
      BEGIN
         SELECT INTO amount_left amount FROM Cash LIMIT 1;
         RETURN (amount_left - $1) >= 10000;
      END;'
     LANGUAGE PLPGSQL;

CREATE OR REPLACE FUNCTION check_money() RETURNS TRIGGER
    AS 'BEGIN
          IF NOT enough_money(NEW.value) THEN
            RAISE EXCEPTION ''Not enough money for this order'';
          END IF;
          RETURN NEW;
        END;'
    LANGUAGE PLPGSQL;

DROP TRIGGER check_money ON Orders;
CREATE TRIGGER check_money
  BEFORE INSERT ON Orders
  FOR EACH ROW
  EXECUTE PROCEDURE check_money();
```

et on peut les utiliser :

```
essais=> CREATE TABLE Cash (amount INTEGER);
CREATE TABLE
essais=> CREATE TABLE Orders (value INTEGER);
CREATE TABLE
essais=> \i mychecks.sql
CREATE FUNCTION
CREATE FUNCTION
DROP TRIGGER
CREATE TRIGGER
```

```
essais=> SELECT amount FROM Cash;
 amount
--------
  30000
(1 row)
essais=> INSERT INTO Orders VALUES (30000);
ERROR:  Not enough money for this order
essais=> INSERT INTO Orders VALUES (20000);
INSERT 372829 1
```

Tout ne peut pas s'exprimer avec ces règles, notamment si on a des règles « métier » très complexes. On doit alors passer aux langages de programmation classiques. Une autre limite, avec PostgreSQL, est que les déclencheurs sont par instruction SQL, pas par transaction (on ne peut pas définir de déclencheur `ON COMMIT`). Si la contrainte d'intégrité s'étend sur plusieurs tables, il n'y a pas de solution.

Une autre limite des règles d'intégrité dans la base est qu'on souhaite parfois mettre ces mêmes règles à d'autres endroits, par exemple dans un code Javascript exécuté sur le navigateur Web du client, afin de lui fournir un rapide retour s'il tape des données erronnées. Cela mène donc à une certaine duplication du code (dans la base et dans le code Javascript) et il faut donc bien veiller à ce que les deux codes demeurent synchrones.

Pour conclure, disons que les règles d'intégrité mises dans base elle-même sont un excellent outil (même avec leurs limites) et que leur utilisation épargnerait bien des ennuis à beaucoup de DBA.

Parmi les articles sur ce sujet, je recommande chaudement "*Constraint Yourself!*" `<http://www.simple-talk.com/sql/t-sql-programming/constraint-yourself!/>` qui donne en outre une bonne liste des techniques possibles.