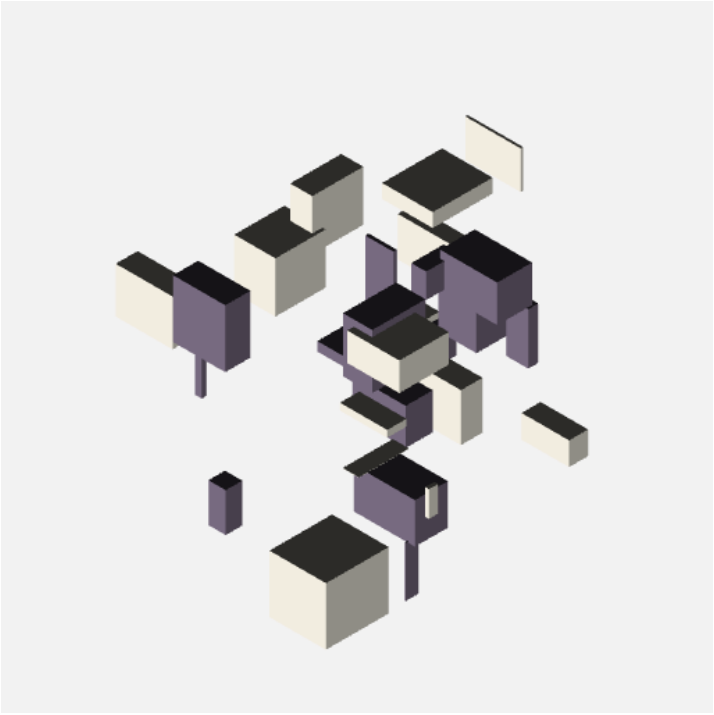
DB2 SQL
Pour développeurs IBMi
Notions de base



Sommaire

Préambule

Ce document est un ancien support de cours que je mets à disposition sous licence Creative Commons.

Il présente des notions de bases qu'il est nécessaire de connaître pour démarrer dans de bonnes conditions sur DB2 pour IBM i. Il met aussi l'accent sur l'utilisation de SQL embarqué dans du code RPG et Adelia.

Ce document couvre en partie les nouveautés apparues sur serveur IBM i à partir de la V7 (et en particulier la V7R1). Pour une présentation plus exhaustive des nouveautés apparues à partir de la V7, on se reportera sur le document « SQL_DBTwo_NewsV7 » qui se trouve dans le même dépôt Github.
Si vous débutez sur DB2 SQL, en particulier sur serveur IBM i, je vous recommande de lire d'abord le document « SQL_DBTwo_Quickstart » qui se trouve aussi dans le même dépôt.

Liens utiles :
https://developer.ibm.com/
https://www.foothing.net/

1. Introduction

Définition Wikipédia :

SQL (sigle de Structured Query Language, en français langage de requête structurée) est un langage informatique normalisé servant à exploiter des bases de données relationnelles. La partie langage de manipulation des données de SQL permet de rechercher, d'ajouter, de modifier ou de supprimer des données dans les bases de données relationnelles.

Outre le langage de manipulation des données :

- le langage de définition des données permet de créer et de modifier l'organisation des données dans la base de données,
- le langage de contrôle de transaction permet de commencer et de terminer des transactions,
- le langage de contrôle des données permet d'autoriser ou d'interdire l'accès à certaines données à certaines personnes.

Source : https://fr.wikipedia.org/wiki/Structured_Query_Language

SQL a quelques caractéristiques intéressants à souligner, qui le distinguent d'autres langages de programmation :

- Langage d'accès aux bases de données relationnelles
- relativement facile à écrire, à lire, à comprendre
- fonctions relationnelles
- langage assertionnel (il se focalise sur le QUOI plutôt que sur le COMMENT)

Les serveurs IBM i (ex AS400) sont généralement utilisés avec le système d'exploitation OS/400, qui a son propre vocabulaire, légèrement différent de celui de SQL. Le tableau de correspondance ci-dessous a pour objet de vous aider à vous y rettrouver.

OS/400	***SQL***
Bibliothèque	Collection / Schema / Database
Fichier physique	Table
Enregistrement	Ligne (row)
Zone	Colonne (column)
Fichier logique sans clé	Vue (view)
Fichier logique avec clé	Index

ATTENTION : si les vues et indexs SQL sont apparentés à des fichiers logiques OS/400, ils ne fonctionnent pas exactement comme des fichiers logiques traditionnels (créés par

l'intermédiaire de DDS).

DML – Data Manipulation Language
SELECT, UPDATE, INSERT, DELETE

DDL – Data Definition Language
CREATE, ALTER, DROP, RENAME
COMMENT ON, LABEL ON

Sécurité/Intégrité
GRANT, REVOKE
LOCK
COMMIT, ROLLBACK, SET TRANSACTION

Procédures, Fonctions, Déclencheurs
BEGIN, END
DECLARE, SET
WHEN, GOTO, WHILE, FOR, ...

Exécution (procédures cataloguées)
CALL

Accès à distance
CONNECT, RELEASE, DISCONNECT
SET CONNECTION

2. Base de données Sample

Depuis la V5R1, l'OS/400 dispose en standard d'une procédure cataloguée permettant la création d'une base de données exemple. Cette procédure cataloguée alimente également un jeu de données dans les tables de cette base de données exemple. Cette procédure crée le schéma spécifié dans le CALL de la procédure. L'appel de la procédure se fait, **à partir d'une session SQL**, de la façon suivante :

```
CALL QSYS/CREATE_SQL_SAMPLE('EXEMPLE')
```

où EXEMPLE est la bibliothèque (ou "schema" en langage SQL) de la base de données exemple créée.

3. Syntaxe générale

Les principaux mots clés utilisés dans le DML (Data Manipulation Language) sont :

`SELECT ...`	colonnes et/ou expressions
`  FROM ...`	quelle(s) table(s) ou vue(s) ?
`  WHERE ...`	quelle(s) condition(s) ?
`  GROUP BY ...`	quel(s) critère(s) de récap ?
`  HAVING ...`	quel(s) groupe(s) ?
`  ORDER BY ...`	quelle séquence ?
`  FETCH FIRST nn ROWS ONLY`	nombre maximum de lignes

Exemple :

```
SELECT * FROM tabempl
```

Exemple avec WHERE :

```
SELECT nom, srv, sal FROM tabempl
  WHERE sal > 14000 ORDER BY srv
```

Exemple avec ORDER BY :

```
SELECT * FROM tabempl ORDER BY dat_nai
```

exemple d'ORDER BY avec n° de colonne (sélection sur 6ème colonne) :

```
SELECT * FROM tabempl ORDER BY 6
```

N.B. : depuis la V5R3, il est possible de faire un ORDER BY sur l'alias d'une zone :

```
SELECT COUNT(*) AS NOMBRE
  FROM COMMANDES
  GROUP BY NUMCLI
  ORDER BY NOMBRE DESC
```

La même chose en utilisant le numéro de la colonne dans la requête.

```
SELECT COUNT(*) AS NOMBRE
  FROM COMMANDES
  GROUP BY NUMCLI
  ORDER BY 1 DESC
```

Il est intéressant de noter que l'on peut insérer un CASE WHEN dans un ORDER BY. Dans l'exemple ci-dessous, les commandes sont triées sur NUMCDE si MODCDE = 'L', et sur DATCDE dans les autres cas :

```
SELECT *
FROM COMMANDES
ORDER BY CASE WHEN MODCDE = 'L' THEN NUMCDE ELSE DATCDE END
```

Exemple avec le prédicat IN :

```
SELECT * FROM tabempl WHERE srv IN (901, 911, 977)
```

Exemple avec le prédicat BETWEEN :

```
SELECT  * FROM tabempl WHERE sal BETWEEN 13200 AND 14500
```

Attention : la manière d'utiliser le prédicat BETWEEN peut avoir un impact sur les performances, notamment dans le cas de comparaison avec des variables hôtes. En règle générale, BETWEEN est utilisée pour comparer une colonne à 2 valeurs en utilisant des variables hôtes de la façon suivante :

```
WHERE COLUMN1 BETWEEN :HOST-VAR1 AND :HOST-VAR2
```

Cependant, il est possible d'utiliser BETWEEN pour comparer une valeur à 2 colonnes de la façon suivante :

```
WHERE :HOST-VAR BETWEEN COLUMN1 AND COLUMN2
```

Mais il est préférable de réécrire la condition ci-dessus de la façon suivante :

```
WHERE :HOST_VAR >= COLUMN1 and :HOST-VAR <= COLUMN2
```

La raison de cette exception est que la formule consistant à comparer une variable hôte à 2 colonnes est moins performante que la formule inverse.

Il faut également noter qu'il est très important de veiller à ce que la première valeur de comparaison du prédicat BETWEEN soit bien inférieure à la seconde valeur. Par exemple :

```
BETWEEN val_mini AND val_maxi
```

Si les valeurs étaient inversées, la requête renverrait un résultat imprévisible.

Exemple avec la clause FETCH nn ROWS ONLY

pour obtenir les 3 première lignes :

```
SELECT * FROM tabempl ORDER BY dat_nai DESC
  FETCH FIRST 3 ROWS ONLY
```

pour obtenir une seule ligne :

```
SELECT * FROM tabempl ORDER BY dat_nai DESC
  FETCH FIRST ROW ONLY
```

Exemple de conditions multiples avec AND et OR :

```
SELECT * FROM tabempl
  WHERE (srv = 977 OR srv = 990) AND sal BETWEEN 13200 AND 14500
  ORDER BY srv DESC, nom
```

Exemple avec le prédicat LIKE

LIKE %

```
SELECT nom, sal FROM tabempl WHERE nom LIKE '%C%'
```

N.B. : on peut recherche le caractère « % » grâce à la clause ESCAPE :

```
... WHERE libelle LIKE '%+%%' ESCAPE '+'
```

LIKE _ (joker)

```
SELECT * FROM tabempl WHERE nom LIKE '_ _N%'
```

N.B. : on peut recherche le caractère « _ » grâce à la clause ESCAPE :

```
... WHERE libelle LIKE 'ZONE!_%' ESCAPE '!'
```

Exemple avec test de valeur indéfinie

```
SELECT * FROM tabempl WHERE srv IS NULL
```

Opérateurs logiques :

=	<	>	<>	<=	>=

OR	
AND	
IN	NOT IN
BETWEEN	NOT BETWEEN
LIKE	NOT LIKE
IS NULL	IS NOT NULL

Opérateurs arithmétiques : + - * / ( )

Exemple avec la clause DISTINCT

Permet la suppression des lignes en double.
Doit suivre immédiatement l'ordre SELECT.

Exemple :

```
SELECT DISTINCT sx FROM tabempl
```

Autre exemple : pour ne pas afficher de lignes identiques éventuelles

```
SELECT DISTINCT * FROM tabempl
```

4. Tables pivot

Une table pivot est une table contenant une seule ligne. Elle est généralement utilisée pour récupérer des donnes du registre DB2, comme par exemple la date système.

Sur la plateforme IBM i, plusieurs tables peuvent être utilisées comme tables pivot, les plus utilisées étant QSQPTABL (qui se trouve dans QSYS2) et SYSDUMMY1 (qui se trouve dans SYSIBM).

Attention : la table QSQPTABL est une table spécifique à l'AS/400, pour une meilleure portabilité du code, il est préférable d'utiliser SYSDUMMY1 qui existe aussi sur DB2 pour LUW (Linux Unix Windows).

Les 2 requêtes ci-dessous sont strictement équivalentes :

```
SELECT CURRENT TIME, CURRENT TIMEZONE FROM SYSIBM.SYSDUMMY1
SELECT CURRENT TIME, CURRENT TIMEZONE FROM QSQPTABL
```

On trouvera dans la suite de ce cours de nombreux exemples d'utilisation des tables pivot.

ATTENTION : ne pas utiliser de table pivot improvisée, en se basant sur une table existante et en utilisant une condition « insensée » (les anglais utilisent le terme « nonsensical query »), car le résultat renvoyé par cette requête sera erroné, et les performances peuvent se révéler désastreuses, si la table utilisée contient beaucoup de lignes. Exemple de requête « insensée » :

```
SELECT CURRENT DATE FROM table WHERE 1 = 0
```

5. Différentes façons d'utiliser SQL

Il existe plusieurs manières d'utiliser SQL en environnement IBMi.

1 – STRQSH

Le principe consiste à exécuter des requêtes SQL dans un programme CL via le SHELL Unix.

Avantage :

- on peut constituer des requêtes par concaténation de variables et ainsi adapter en « live » une requête au contexte du traitement.

Inconvénients :

- syntaxe difficile à mettre au point en cas de concaténation de plusieurs paramètres
- la syntaxe SQL utilisée avec STRQSH est impérativement la syntaxe SQL de norme ISO, et non la syntaxe SQL IBM i, le caractère séparateur entre bibliothèque et fichier est donc le point et non le slash.
- chaque instruction DB2 appelée via STRQSH s'exécute dans sa propre session, et ne « voit » pas les fichiers temporaires qui ont put être constitués par d'autres requêtes. Ce principe limite quelque peu l'utilisation

Exemples d'utilisation :

```
STRQSH CMD('db2 "SELECT BACDOGA, BANOPRT FROM MYLIBRARY.prtp
 WHERE BACDOGA = ''ATL'' "')

STRQSH CMD('db2 "CREATE TABLE MYLIBRARY.X_TCTAFFP AS
 ( SELECT * FROM MYLIBRARY.TCTAFFP) WITH DATA"')
```

Autre exemple d'utilisation :

```
DCL         VAR(&COTE) TYPE(*CHAR) LEN(1) VALUE('''')
DCL         VAR(&REQSQL) TYPE(*CHAR) LEN(100) VALUE('db2 +
            "UPDATE CASTTOOL.SCRIPTFTP SET ZGET = +
            REPLACE(ZGET, ''BIBREFXXX'', ''')
CHGVAR      VAR(&REQSQL) VALUE(&REQSQL *TCAT +
            &BIBREFORI *TCAT &COTE *CAT ')"')
STRQSH      CMD(&REQSQL)
```

=> permet d'obtenir la requête ci-dessous dans le cas où &BIBREFORI = 'A440DSRC' :

```
db2 "UPDATE CASTTOOL.SCRIPTFTP
  SET ZGET = REPLACE(ZGET, 'BIBREFXXX', 'A440DSRC')"
```

2 – RUNSQLSTM

Le principe consiste à stocker les requêtes dans un membre de fichier source, et à exécuter les requêtes se trouvant dans le fichier source via la commande RUNSQLSTM.

Avantage :

- tous les types de requêtes sont permis (SELECT, UPDATE, DELETE, DROP, CREATE TABLE, etc…)
- possibilité d'enchaîner plusieurs requêtes dans un même fichier source,
- possibilité de choisir entre plusieurs modes de contrôle de validation,
- possibilité de choisir entre l'appellation SQL (chemin d'accès avec point en caractère séparateur) ou l'appellation système (chemin d'accès avec slash en caractère séparateur).

Inconvénients :

- pas de possibilité de passer des paramètres aux requêtes à exécuter.
- impossibilité d'utiliser le !! pour les concaténations de chaînes (syntaxe spécifique IBM i). Mais on peut contourner le problème en utilisant l'ordre SQL CONCAT qui est strictement équivalent.

Exemple d'utilisation :

```
RUNSQLSTM  SRCFILE(&BIB/QSQLSRC) SRCMBR(SVPR01SQL) COMMIT(*NONE)
```

Contenu du fichier source &BIB/QSQLSRC, membre SVPR01SQL :

```
----------------------------------------
CREATE TABLE MYLIBRARY/X_TCTAFFP AS
( SELECT *
  FROM TCTAFFP
) WITH DATA ;
----------------------------------------
CREATE TABLE MYLIBRARY/X_TTYPAFP AS
( SELECT *
  FROM TTYPAFP
) WITH DATA ;
----------------------------------------
```

3 – STRQMQRY

Le principe consiste à stocker les requêtes SQL dans des objets de type QM, et à les exécuter via la commande STRQMQRY.

Avantages :

- possibilité de définir des formats d'impression sophistiqués de type query
- possibilité de passer des paramètres grâce au paramètre SETVAR
- possibilité de générer des fichiers à partir de requêtes SELECT grace au paramètre OUTFILE
- possibilité de déployer les requêtes plus facilement sur une machine cible, qu'à partir de fichiers sources, car chaque requête QM est contenue dans un objet facilement copiable et/ou déplaçable.

Inconvénients :

- pas de possibilité de faire d'autres types de requêtes que SELECT
- une seule requête peut être exécutée à l'intérieur d'un objet de type QM

Exemple d'utilisation :

```
STRQMQRY    QMQRY(TEST/DOSIPEAU) OUTPUT(*OUTFILE) +
              OUTFILE(QTEMP/PRXFLU) SETVAR((NOMCIBLE +
              &NOMCIBLE))
```

Requête QM TEST/DOSIPEAU :

```
SELECT RACDOGA, RACDTTYCPT, RAACTIVITE, RACDPAT1, RACDPAT2, RACDPAT3,
  RACDPAT4, RADTDBVL, RAPRXUNTFL
FROM priprvp A, &NOMCIBLE B
WHERE
 A.RACDOGA!!A.RAACTIVITE!!A.RACDPAT1 =
 B.NDCDOGA!!B.NDACTIVITE!!B.NDCDPAT1
```

4 – RUNQRY

Le principe consiste à stocker les requêtes dans des objets de type QUERY, et à les exécuter via la commande RUNQRY. C'est sans doute la méthode la plus ancienne, et elle est encore beaucoup pratiquée. Ne permet que des requêtes de type SELECT (avec néanmoins la possibilité de créer des fichiers DB2/400). Ne permet pas le passage de paramètres (même si on peut contourner le problème en effectuant une jointure avec une table contenant des valeurs paramétrées).

5 – Autres modes d'utilisation possibles

Les modes d'utilisation listés ci-dessous ne sont pas spécifiques au langage CLP, mais surtout ils permettent de lever un grand nombre des restrictions vues dans les chapitres précédents :

- à l'intérieur d'une procédure cataloguée,
- à l'intérieur d'une UDF (User Defined Function) ou d'une UDTF (User Defined Table Function),
- à l'intérieur d'un programme SQLRPGLE (cf. chapitre « SQL et RPG").

6 – Appeler un programme ou une commande OS/400 via SQL

Il est possible d'exécuter sous SQL une commande AS400, ou un programme CLP, via l'API QCMDEXC.

Il suffit de passer à l'API QCMDEXC les paramètres suivants :
- Une chaîne de caractères contenant la commande à exécuter.
- La longueur de la commande à exécuter (décimal de 10,5).

L'exemple ci-dessous met en oeuvre un OVRDBF sur un fichier, puis l'enlève :

```
C/Exec SQL
C+ CALL QCMDEXC('OVRDBF FILE(TOTO1) TOFILE(TOTO2)', 32)
C/End-Exec
```

```
  Exec SQL
     CALL QCMDEXC('DLTOVR FILE(*ALL)', 17);
```

Dans l'exemple ci-dessus, l'OVRDBF est écrit en utilisant l'ancienne écriture SQL RPG au format fixe, et le DLTOVR est écrit en utilisant la technique équivalente pour le RPG Free.

A partir de la V7R1, on n'est plus obligé de préciser la longueur de la commande à exécuter, et on peut donc écrire ceci :

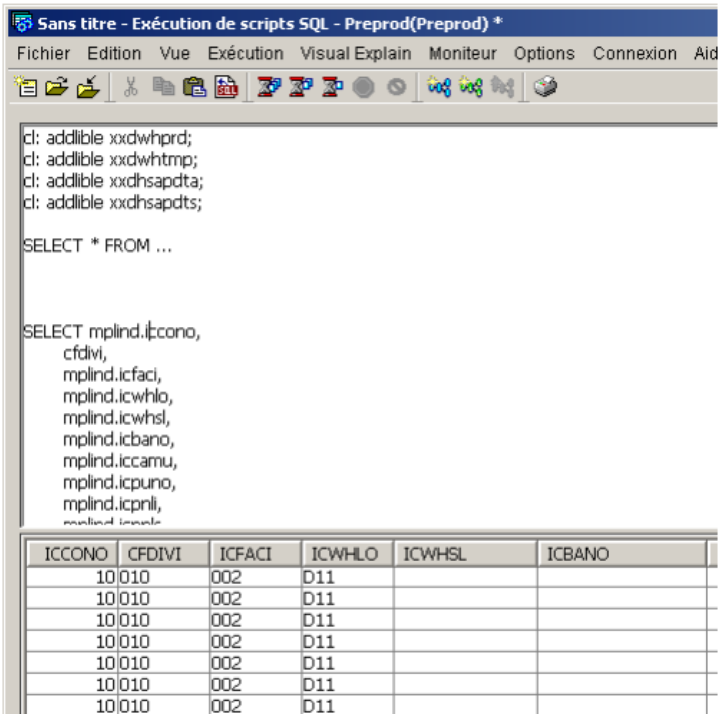
```
  Exec SQL
     CALL QCMDEXC('DLTOVR FILE(*ALL)');
```

A noter : cette technique peut être utilisée également à l'intérieur de procédures stockées, par exemple pour effectuer un CLRPFM (généralement plus rapide qu'un DELETE SQL), mais il convient d'être prudent avec les objets déclarés avec des noms longs (supérieurs à 10 caractères) car le CLRPFM ne sait "travailler" qu'avec les noms courts (cf. cours chapitre dédié à ce sujet dans le cours SQL Avancé) :

```
CALL QCMDEXC ('CLRPFM bib/table', 16) ;
```

7 – A l'intérieur d'une fenêtre IBM i Navigator

Il est souvent nécessaire de forcer – en préalable à l'exécution de requêtes - une liste de bibliothèques permettant de se mettre dans le contexte d'un travail IBM i équivalent. Pour ce faire, utiliser la commande « cl » de la façon suivante :



Pour pouvoir utiliser le principe des listes de bibliothèques, qui est beaucoup utilisé en environnement IBMi, il faut au préalable se mettre en mode « syntaxe SQL ». Cette option se situe dans le menu :

Connexion -> Sous-menu "paramètres JDBC" -> onglet "Format" -> option "Convention d'appellation" qui doit être à *SYS

Avant la V7R1, le slash séparateur entre bibliothèque et table était obligatoire avec la convention d'appelletion *SYS. Depuis la V7R1, ce n'est plus le cas, on peut donc utiliser indifféremment le point ou le slash.

La convention d'appellation "SQL" correspond à la syntaxe SQL ISO, elle n'accepte pas de notion de liste de bibliothèque. Le slash de séparation entre bibliothèque et table n'est pas toléré dans ce mode et doit être remplacé par le point.

8 – STRSQL

C'est l'équivalent de la fenêtre d'exécution de code SQL de System i Navigator (vue au point précédent), mais en mode 5250.
Cette solution doit être vue comme une solution de dépannage. IBM ne la fait plus évoluer, et elle est beaucoup moins pratique qu'un client SQL en mode graphique comme System i Navigator.

9 - Client SQL graphique alternatif

De nombreux clients SQL en mode graphique existent. Certains sont gratuits et open source, comme SquirrelSQL, d'autres sont des solutions payantes (comme WinSQL). Ils attaquent généralement la base de données DB2 for i via JDBC et un driver IBMi complémentaire. Dans le cas de SquirrelSQL, ce dernier "attaque" la base de données DB2 for i via JTOpen400, un driver dédié à DB2 for i pouvant être utilisé avec différents logiciels Java s'appuyant sur JDBC.

Lien vers SquirrelSQL :
http://squirrel-sql.sourceforge.net/

Lien vers JTOpen400 :
http://jt400.sourceforge.net/

Ces clients SQL alternatifs offrent généralement la possibilité de travailler sur plusieurs bases de données (DB2 for i, DB2 pour LUW, Oracle, MySQL, SQLServer, etc..) au traver d'une seule interface. C'est notamment le cas de SquirrelSQL.

Exemple de chaîne de connexion (en anglais : "connexion string") pour établir une connexion avec JTOpen400 sur une base DB2 for i :

jdbc:odbc:DRIVER={iSeries Access ODBC Driver};SYSTEM=PREPROD;

Exemple de chaîne de connexion pour établir une connexion avec le driver spécifique à la base DB2 for LUW (Linux Unix Windows) :

jdbc:odbc:Driver={IBM DB2 ODBC
DRIVER};Hostname=localhost;Port=50000;Protocol=TCPIP;Database=DB2SAMPL;

Site très utile pour connaître la syntaxe des "connexions strings", pour les principales bases de données du marché (dont DB2) :
http://www.connectionstrings.com/

10 – RUNSQL

Apparue sur la V7R1 de l'IBM i, la commande RUNSQL est extrêmement pratique.
Elle permet d'exécuter une instruction SQL depuis un programme CL sans avoir besoin d'un fichier source.

Les instructions suivantes peuvent être lancées avec RUNSQL :

ALTER FUNCTION

ALTER PROCEDURE

ALTER SEQUENCE

ALTER TABLE

CALL

COMMENT

COMMIT

CREATE ALIAS

CREATE FUNCTION

CREATE INDEX

CREATE PROCEDURE

CREATE SCHEMA

CREATE SEQUENCE

CREATE TABLE

CREATE TRIGGER

CREATE TYPE

CREATE VARIABLE

CREATE VIEW

DECLARE GLOBAL TEMPORARY TABLE

DELETE

DROP

GRANT

INSERT

LABEL

MERGE

REFRESH TABLE

RELEASE SAVEPOINT

RENAME

REVOKE

ROLLBACK

SAVEPOINT

SET CURRENT DECFLOAT ROUNDING MODE

SET CURRENT DEGREE

SET CURRENT IMPLICIT XMLPARSE OPTION

SET ENCRYPTION PASSWORD

SET PATH

SET SCHEMA

SET TRANSACTION

UPDATE

Documentation officielle :
https://www.ibm.com/docs/en/i/7.1?topic=environments-using-runsql-cl-command

La chaîne d'instruction peut compter jusqu'à 5000 caractères. Elle ne doit pas se terminer par un point-virgule.

Les commentaires sont autorisés dans la chaîne d'instruction. Un commentaire de ligne commence par un trait d'union double (--) et se termine à la fin de la ligne (retour de ligne et/ou alimentation de ligne) ou à la fin de la chaîne. Les commentaires de bloc commencent par /* et continuent jusqu'à ce que le */ correspondant soit atteint. Les commentaires de bloc peuvent être imbriqués.

Si un fichier est ouvert par RUNSQL, il est fermé avant que le contrôle ne soit retourné à l'appelant. Si le contrôle d'engagement est actif, il appartient à l'application de l'utilisateur d'effectuer le commit ou le rollback.

La commande s'exécute dans le groupe d'activation de l'invocateur. Si RUNSQL est inclus dans un programme CL compilé, le groupe d'activation du programme est utilisé.

Aucune liste de sortie n'est générée. En cas d'échec, le message SQL est envoyé comme message d'échappement à l'appelant. Pour une instruction SQL complexe qui retourne une erreur de syntaxe, vous pouvez utiliser le moniteur de base de données pour aider à trouver la cause de l'erreur. Démarrer un moniteur de base de données, exécuter la commande RUNSQL et analyser le moniteur de base de données à l'aide de System i® Navigator.

Exemple avec l'ordre SQL INSERT :

```
RUNSQL SQL('INSERT INTO prodLib/work_table VALUES(1, CURRENT TIMESTAMP)')
```

Dans un programme CL, vous pouvez utiliser la commande Receive File (RCVF) pour lire les résultats de la table générée pour cette requête : :

```
RUNSQL SQL('CREATE TABLE qtemp.worktable AS
  (SELECT * FROM qsys2.systables WHERE table_schema = ''MYSCHEMA'') WITH DATA')
       COMMIT(*NONE) NAMING(*SQL)
```

Exemple de CL avec construction dynamique d'une requête SQL (en fonction d'un paramètre reçu) et exécution :

```
RUNSQL1: PGM PARM(&LIB)
    DCL &LIB TYPE(*CHAR) LEN(10)
    DCL &SQLSTMT TYPE(*CHAR) LEN(1000)
    CHGVAR VAR(&SQLSTMT) +
           VALUE('DELETE FROM qtemp.worktable1 +
                  WHERE table_schema = ''' || &LIB || ''')
    RUNSQL SQL(&SQLSTMT) COMMIT(*NONE) NAMING(*SQL)
RUNSQL1: ENDPGM
```

6. Création d'une base DB2

6.1 Schema

Les objets créés par l'intermédiaire de SQL (tables, vues, etc...) peuvent être placés :

- dans une bibliothèque AS/400 « normale »
- dans une collection elle-même créée par SQL

Les tables créées dans une collection seront journalisées automatiquement

Remarque : SCHEMA est synonyme de COLLECTION et de DATABASE.

Création d'une collection SQL :

```
CREATE COLLECTION personnel ;
```

Cette instruction créera la bibliothèque « personnel », de type « *PROD » avec TEXT('Collection créée par SQL'), et contenant :

- un récepteur de journal QSQJRN0001 avec THRESHOLD(1441000) (=> seuil d'alerte fixée à 1,4 Go)
- un journal QSQJRN avec MNGRCV(*SYSTEM) et RCVSIZOPT(*RMVINTENT *MAXOPT2)
- un catalogue SQL

Une collection peut :
résider dans un ASP (IN ASP n)
inclure un dictionnaire IDDU (WITH DATA DICTIONARY)

Le Catalogue

- il est composé d'un ensemble de vues SQL pointant sur les fichiers QADB* de QSYS (références croisées DB2 de l'OS/400)
- il contient les informations concernant les objets de la collection
- il est créé et maintenu automatiquement (il ne peut être supprimé ou modifié explicitement)
- il peut être consulté, par exemple par SELECT

Remarque : un catalogue général de tout DB2 est maintenu dans QSYS2. Ce catalogue comporte également quelques tables complémentaires. En V5R2, des catalogues ODBC/JDBC et ANSI/ISO sont aussi fournis dans SYSIBM.

6.2 Tables

Création d'une table DB2

```
CREATE TABLE tabnouv
 (mat   SMALLINT       NOT NULL,
  nom   CHAR(20)       NOT NULL,
  cat   DEC(1),
  sexe  CHAR(1)        NOT NULL,
  srv   SMALLINT       NOT NULL WITH DEFAULT 999,
  dataj TIMESTAMP      NOT NULL WITH DEFAULT
);
```

Lors de l'ajout d'une ligne :
- NOT NULL : donnée obligatoirement fournie
- NOT NULL WITH DEFAULT <valeur ou USER> :
  - si donnée non fournie et pas de valeur explicite
    - espaces, zéros, CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP
  - si donnée non fournie et valeur explicite
    - cete valeur (USER = nom du profil)
- Pas d'indication :
  - si donnée non fournie : valeur indéfinie (NULL)

Particularités des tables

Une table est un fichier physique étiqueté :
- type de fichier SQL = TABLE
- taille du membre (SIZE) == *NOMAX
- réutilisation enregs supprimés (REUSEDLT) = *YES
- nom du format = nom de la table

Ce fichier est journalisé, si le journal QSQJRN existe dans la bibliothèque de création (cas de la collection), avec :
- images journal (IMAGES) = *BOTH
- postes à ométtre (OMTJRNE) = *OPNCLO

Noms longs et noms courts

Un **nom de table** SQL peut être créée avec un nom court (longueur maximum <= 10 caractères) ou un nom long (de longueur comprise entre 11 et 128 caractères). Les tables créées avec un nom long se voient attribuer automatique un nom court par le système, si ce nom long dépasse 10 caractères. Ce nom court est constitué à partir des 5 premiers caractères du nom long, suivi d'un compteur de 5 chiffres attribué par le système. Le nom long est un alias pointant sur le nom court. Le nom court permet d'utiliser la table dans les programmes RPG et les querys. SQL permet d'utiliser indifféremment les noms courts et les noms longs.

Un **nom de colonne** peut être créé avec un nom court (longueur maximum <= 10 caractères) ou un nom long (de longueur comprise entre 11 et 30 caractères). Les colonnes créées avec un nom long se voient attribuer automatique un nom court par le système. Ce nom court est constitué à partir des 5 premiers caractères du nom long, suivi d'un compteur de 5 chiffres attribué par le système. Comme pour les noms de tables, le nom long d'une colonne est un alias pointant sur le nom court. Le nom court permet d'utiliser la colonne dans les programmes RPG et les querys. SQL permet d'utiliser indifféremment les noms courts et les noms longs.

Exemple de table créée avec des noms de tables et de colonnes longs :

```
CREATE TABLE supplier_test (
supplier_name  CHAR(30),
supplier_addr1 CHAR(30),
supplier_phone CHAR(20)
);
```

nom de la table:

Nom long	Nom court
SUPPLIER_TEST	SUPPL00001

noms des colonnes de la table :

Nom long	Nom court	Type
supplier_name	SUPPL00001	char(30)
supplier_addr1	SUPPL00002	char(30)
supplier_phone	SUPPL00003	char(20)

On constate que les noms système sont abscons, et difficiles à utiliser avec RPG et Query.

Pour obtenir un nom court de table significatif, on peut forcer un nom court de son choix au moyen de la commande suivante :

```
RENAME TABLE supplier_test TO SYSTEM NAME suppltest ;
```

Pour obtenir des noms courts de colonnes significatifs et lisibles en regard des noms longs, on peut spécifier la clause FOR COLUMN lors de la définition de colonnes dont les noms dépassent

10 caractères :

```
create table supplier (
supplier_name  for column suppname char(30),
supplier_addr1 for column suppaddr1 char(30),
supplier_phone for column suppphone char(20),
... )
```

Le nom système doit être différent du nom de colonne, aussi faut-il omettre la clause FOR COLUMN sur les colonnes ayant déjà des noms courts, sauf si l'on veut supprimer des caractères tels que le trait de soulignement.

Petite astuce pour ceux qui créent des tables via DDS, mais qui souhaitent pouvoir utiliser des noms longs avec SQL : le mot-clé ALIAS permet de spécifier pour chaque champ un nom de colonne SQL différent du nom de champ DDS :

```
D SUPPNAME  ALIAS SUPPLIER_NAME )
D SUPPADDR1 ALIAS SUPPLIER_ADDR1 )
D SUPPPHONE ALIAS SUPPLIER_PHONE )
```

CLE PRIMAIRE

- une seule par table
- assure l'unicité de chaque ligne
- NOT NULL obligatoire
- Par CREATE/ALTER TABLE

UNICITE

- Null admis
- Par CREATE TABLE/ALTER TABLE ou CREATE UNIQUE INDEX

Exemples :

```
CREATE TABLE bidon
  (a CHAR(10) NOT NULL, PRIMARY KEY(a));

ALTER TABLE tabempl ADD UNIQUE(mat);
```

ATTRIBUT IDENTITY

- pour l'incrémentation automatique d'un identifiant numérique (0 déc)

Exemple :

```
CREATE TABLE incrauto (
   numero INTEGER
      GENERATED ALWAYS AS IDENTITY
      (START WITH 10, INCREMENT BY 10),
   libelle CHAR(15))
INSERT INTO incrauto (libelle) VALUES('azertyuiop')
SELECT * FROM incrauto ;
```

INTEGRITE REFERENTIELLE

Par ALTER TABLE, exemples :

```
ALTER TABLE tabempl ADD
  FOREIGN KEY(srv) REFERENCES tabserv(srv)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION ;
```

Valeurs possibles pour ON DELETE

- NO ACTION (défaut)
- CASCADE
- RESTRICT
- SET DEFAULT
- SET NULL

Valeurs possibles pour ON UPDATE :

- NO ACTION (défaut)
- RESTRICT

Sur CREATE TABLE , exemple :

```
CREATE TABLE coll/test (colonnes,
  CONSTRAINT exemple_cours
  FOREIGN KEY (mat)
  REFERENCES tabempl (mat)
  ON DELETE CASCADE
  ON UPDATE RESTRICT);
```

L'intégrité référentielle nécessite la mise en oeuvre de la journalisation, sauf dans le cas d'un ON DELETE RESTRICT ON UPDATE RESTRICT.

CREATE TABLE (RCDFMT)

Depuis la V5R4, le CREATE TABLE admet un nouveau paramètre RCDFMT, permettant d'indiquer un nom de format :

Exemple :

```
CREATE TABLE client (
   nocli INTEGER,
   nom CHAR(35)
) RCDFMT clientf ;
```

CONTRAINTE DE VERIFICATION – La clause CHECK

Permet d'appliquer un contrôle de validité sur certaines colonnes lors des INSERT/UPDATE
L'ajout de contrainte peut se faire par CREATE et ALTER TABLE.
La syntaxe de la contrainte est la même que celle utilisable dans un WHERE.

Exemples :

```
CREATE TABLE mabib/service(
  rqid  SMALLINT NOT NULL
          CONSTRAINT  rqid_pk PRIMARY KEY,
   status VARCHAR(10) NOT NULL
       WITH DEFAULT 'NEW'
       CHECK ( status IN ( 'NEW', 'ASSIGNED', 'Pending', 'CANCELLED' )  ),
   rq_desktop CHAR(1) NOT NULL
       WITH DEFAULT 'N'
       CHECK ( rq_desktop IN ( 'Y', 'N' ) ),
   rq_ipaddress CHAR(1) NOT NULL
       WITH DEFAULT 'N'
       CHECK ( rq_ipaddress IN ( 'Y', 'N' ) ),
   rq_unixid CHAR(1) NOT NULL
       WITH DEFAULT 'N'
       CHECK ( rq_unixid IN ( 'Y', 'N' ) ),
   staffid INTEGER NOT NULL,
   techid  INTEGER,
   accum_rqnum INTEGER NOT NULL
       GENERATED ALWAYS AS IDENTITY
         ( START WITH 1,
         INCREMENT  BY 1,
       CACHE 10 ),
  comment  VARCHAR(100) );

ALTER TABLE tabempl
  ADD CONSTRAINT sal_mini CHECK(sal > 8000) ;
```

LABEL ON COLUMN

Permet d'affecter l'équivalent des mots-clés COLHDG et TEXT

```
LABEL ON COLUMN tabempl
      (nom IS 'Nom employé',
       sx     IS 'Sexe',
       srv    IS 'Service',
       sal    IS 'Salaire');

LABEL ON COLUMN tabempl
      (sx TEXT IS 'Indiquer F ou M');
```

remarque : on peut aussi ajouter un commentaire par COMMENT ON COLUMN

LABEL ON INDEX

Cette fonctionnalité, disponible depuis la V5R4, permet de s'affranchir de la commande CHGOBJD pour affecter un nom à un index.
Exemple :

```
Label On Index PRODL2 Is 'Liste des produits (DATE TYP NUM)'
```

COMMENT ON

- on peut affecter un commentaire aux objets SQL

```
COMMENT ON TABLE tabempl IS 'Société xyz'

COMMENT ON INDEX idxnom IS 'clé nom sur TABEMPL'

COMMENT ON ALIAS employes IS 'Alias pour TABEMPL'

COMMENT ON VIEW ...
```

- les commentaires sont enregistrés dans le catalogue SQL
- on peut aussi ajouter un texte par LABEL ON

CREATE TABLE ... AS ...

Depuis la V5R2, il est possible de créer une table en utilisant une sous-requête :

- pour créer un fichier avec les données résultantes :

```
CREATE TABLE TABTEMP AS
(SELECT codart, libart FROM TABART) WITH NO DATA ;
```

- pour créer un fichier sans les données résultantes :

```
CREATE TABLE TABTEMP as
(SELECT codart, libart FROM TABART) WITH NO DATA ;
```

La clause with no data est prise par défaut si vous ne l'indiquez pas.

CREATE TABLE ... LIKE ...

Création d'une table à partir d'une table DB2, ou d'une vue DB2, sans les données :

```
      CREATE TABLE employes LIKE tabempl ;
```

La technique ci-dessus est finalemnet synonyme de la technique ci-dessous :

```
CREATE TABLE employes AS
( SELECT * FROM tabempl)
WITH NO DATA ;
```

6.3 Indexs

CREATE INDEX

- un index peut être créé pour :
  - assurer l'unicité des clés (UNIQUE INDEX)
  - améliorer les performances des requêtes

- un index est créé exclusivement sur une TABLE (ne fonctionne pas une vue)

```
CREATE INDEX idxsx ON tabempl(sx)

CREATE UNIQUE INDEX idxno ON tabempl(no)

CREATE UNIQUE WHERE NOT NULL INDEX idxid
     ON tabempl(nom, dat_nai DESC)
```

- les 3 instructions précédentes créent un chemin d'accès de type « arbre binaire »

- un index n'est jamais cité explicitement dans une requête, mais il est utilisé en général pour :
  - ORDER BY
  - GROUP BY
  - JOIN
  - WHERE

- l'optimiseur de requête peut créer dynamiquement un index temporaire

- caractéristiques de la clé :
  - nombre maximum de colonnes : 120
  - longueur maximum = 2000 – nombre de colonnes
  - chaque colonne peut être ascendante ou descendante (DESC)

Un index est un fichier logique étiqueté :
- type de fichier SQL : INDEX
- type de chemin d'accès : Par clé
- nom du format : nom de la table

6.4 Vues

CREATE VIEW

- une vue est basée :
  - sur une ou plusieurs TABLES ou VUES
  - avec une sélection de colonnes et/ou de lignes

```
CREATE VIEW srv_977
   AS SELECT nom, srv
   FROM tabempl
   WHERE srv = 977
```

- après création, une vue est utilisable comme une table :

```
SELECT * FROM srv_977
```

- une vue peut être récapitulative :

```
CREATE VIEW srvsal(srv, totsal)
  AS SELECT srv, DECIMAL(SUM(sal), 8, 2)
  FROM tabempl GROUP BY srv
```

- une vue peut être une jointure :

```
CREATE VIEW empserv
  AS SELECT mat, nom, nomsrv
  FROM tabempl JOIN tabserv
  ON tabempl.srv = tabserv.srv
```

Lors de la création de vues, on recommandera l'utilisation de la clause "OR REPLACE", qui a pour effet de recréer la vue si elle existe déjà :

```
CREATE OR REPLACE VIEW empserv
  AS SELECT mat, nom, nomsrv
  FROM tabempl JOIN tabserv
  ON tabempl.srv = tabserv.srv
```

La clause "OR REPLACE" a pour effet que l'objet recréé conserve les caractéristiques (notamment en termes de droits) de la vue telles qu'elles **étaient** définies avant sa recréation.

Particularités des vues :

Une vue est un fichier logique étiqueté :
- type de fichier SQL : VUE
- instruction de création de vue SQL : le CREATE VIEW complet (visible au moyen de la commande IBM i DSPFD)
- type de chemin d'accès : Arrivée
- nom du format : nom de la vue
- texte du format : FORMAT0001

Une vue est assimilable à un fichier logique SANS clé, mais dispose de possibilités beaucoup plus étendues... On peut en effet à la création :
- définir des colonnes calculées
- utiliser des fonctions scalaires
- utiliser GROUP BY et HAVING
- utiliser des sous-requêtes

On ne peut pas lors de la création demander :
- FOR UPDATE OF
- ORDER BY

La vue créée sera en lecture seule si :
- FROM sur plusieurs TABLES et/ou VUES
- FROM sur une VUE elle-même en lecture seule
- DISTINCT
- GROUP BY
- utilisation d'une fonction de colonne

On peut spécifier WITH CHECK OPTION, qui interdira tout ajout ou mise à jour par cette vue qui rendrait la ligne inaccessible par cette même vue.

6.5 Tables temporaires

DECLARE GLOBAL TEMPORARY TABLE

La création de table temporaire au moyen de l'instruction DECLARE GLOBAL TEMPORARY TABLE est apparue en V5R2

Cette technique s'appuie sur les paramètres de session SQL, et donc sur les paramètres du travail AS/400 exploitant cette session. QTEMP étant la bibliothèque temporaire du travail AS/400, elle est implicitement retenue par SQL pour la constitution des tables temporaires.

Elle constitue une alternative intéressante au traditionnel CRTDUPOBJ, ou au CREATE TABLE QTEMP/fichier, en offrant une syntaxe « full SQL » portable d'une plateforme à une autre (on retrouve d'ailleurs cette forme d'écriture dans les procédures stockées).

Elle permet d'utiliser les mêmes clauses WITH DATA, AS et LIKE que le CREATE TABLE.

Elle permet aussi d'utiliser la clause WITH REPLACE qui permet de forcer le remplacement d'une table existant déjà dans QTEMP. Si cette clause n'est pas indiquée, et si la table existe déjà dans QTEMP, la création de la table temporaire échouera.

On peut également utiliser les clauses ON COMMIT et ON ROLLBACK (à compléter).

Exemples :

```
DECLARE GLOBAL TEMPORARY TABLE toto (
     mat integer,
     nom char(12))

DECLARE GLOBAL TEMPORARY TABLE tmpemp LIKE tabempl
    WITH REPLACE

DECLARE GLOBAL TEMPORARY TABLE tmpemp AS
     (SELECT * FROM tabempl)
      WITH DATA  WITH REPLACE
```

Il n'existe pas de directive de type « WITH NO DATA » mais il est possible d'utiliser à la place la directive explicite "DEFINITION ONLY" :

```
DECLARE GLOBAL TEMPORARY TABLE tmpemp AS
     (SELECT * FROM tabempl)
      DEFINITION ONLY WITH REPLACE
```

D'autres paramètres optionnels peuvent être utilisés pour la création de tables temporaires, tels que :

- ON COMMIT [PRESERVE ROWS | DELETE ROWS]
- ON DELETE [PRESERVE ROWS | DELETE ROWS]
- NOT LOGGED

Dans l'exemple ci-dessous, la table temporaire tabtmp est créée par duplication de la table tabref, cette création n'est pas journalisée (NOT LOGGED), donc insensible aux rollbacks. Et en cas de COMMIT, toutes les lignes de la table temporaire seront conservées.

```
DECLARE GLOBAL TEMPORARY TABLE tabtmp
LIKE tabref
ON COMMIT PRESERVE ROWS
NOT LOGGED
```

Si on avait utilisé ON COMMIT DELETE ROWS, cela aurait signifié que l'on souhaitait forcer la suppression de toutes les lignes de la table temporaire à chaque COMMIT.

POINT IMPORTANT : si un ROLLBACK est exécuté à l'intérieur d'une transaction, ou unité de travail (en anglais : UOW = Unit of Work), alors toutes les tables temporaires créées à l'intérieur de cette transaction sont supprimées.

6.6 Modification de tables

ALTER TABLE

L'ordre ALTER TABLE permet de modifier différents attributs d'une table ou de ses colonnes. Le mode SQL interactif propose les options suivantes :

1 = ADD contrainte,
2 = DROP contrainte,
3 = ADD zone,
4 = ALTER zone,
5 = DROP zone.

L'option 1=ADD Contrainte permet d'accéder aux options suivantes :

1 = PRIMARY KEY,
2 = UNIQUE KEY,
3 = FOREIGN KEY,
4 = CHECK.

Exemples de modification d'une table Contacts :

- création préliminaire de la table :

```
CREATE TABLE CONTACTS (
  ID      NUMERIC (10,0) NOT NULL WITH DEFAULT,
  NOM     CHAR (30)      NOT NULL WITH DEFAULT,
  ADRMAIL CHAR (30)      NOT NULL WITH DEFAULT) ;
```

- ajout d'une contrainte de type "clé primaire" sur la colonne ID :

```
ALTER TABLE CONTACTS ADD CONSTRAINT KEY_PRIM_CONTACT PRIMARY KEY (ID);
```

- modification de la taille et du type de la colonne ADRMAIL :

```
ALTER TABLE CONTACTS
  ALTER COLUMN ADRMAIL
    SET DATA TYPE VARCHAR (80) ALLOCATE(20) CCSID 297 NOT NULL WITH DEFAULT ;
```

- création d'une table "Services" pour rattachement ultérieur à la table "Contacts" :

```
CREATE TABLE SERVICES (
   ID     NUMERIC ( 10, 0) NOT NULL WITH DEFAULT,
   NOM    CHAR ( 30)       NOT NULL WITH DEFAULT,
   CONSTRAINT KEY_PRIM_SERV PRIMARY KEY (ID)) ;
```

- ajout d'une colonne "SERVICE_ID" dans la table « Contacts » et pour rattachement comme clé étrangère à la table « Services » :

```
ALTER TABLE CONTACTS
  ADD COLUMN SERVICE_ID NUMERIC (10 , 0) NOT NULL WITH DEFAULT
  CONSTRAINT FOR_KEY_SERV_CONT REFERENCES SERVICES(ID)
     ON DELETE NO ACTION ON UPDATE NO ACTION ;
```

La définition d'une clé étrangère offre les options suivantes sur les options de suppression (on delete) et de mise à jour (on update) :

- ON DELETE :
  - 1 = NO ACTION,
  - 2 = RESTRICT,
  - 3 = CASCADE,
  - 4 = SET NULL
  - 5 = SET DEFAULT
- ON UPDATE :
  - 1 = NO ACTION,
  - 2 = RESTRICT

Si on souhaite modifier la contrainte « FOR_KEY_SERV_CONT » de façon à effectuer une suppression en cascade des contacts liés à un service supprimé, on devra d'abord supprimer cette contrainte, puis la recréer comme dans l'exemple ci-dessous :

```
ALTER TABLE CONTACTS
   DROP FOREIGN KEY FOR_KEY_SERV_CONT CASCADE ;

ALTER TABLE CONTACTS
   ADD CONSTRAINT FOR_KEY_SERV_CONT FOREIGN KEY (SERVICE_ID)
     REFERENCES MABIB/SERVICES (ID)
       ON DELETE CASCADE ON UPDATE NO ACTION ;
```

On rappelle qu'il est possible d'insérer une colonne dans une table existante au moyen d'un ALTER TABLE... ADD COLUMN....
La colonne ajoutée sera toujours placée à la fin de la liste des colonnes de la table altérée, sauf si l'on précise explicitement que l'on souhaite ajouter la nouvelle colonne AVANT une autre colonne au moyen du mot clé BEFORE, ce qui nous donne ceci :

```
ALTER TABLE my_table ADD COLUMN new_column ... BEFORE old_column ;
```

Il n'existe pas sur DB2 for i de possibilité de renommer une colonne, mais on peut détourner la technique précédente pour effectuer ce renommage. La technique consiste à procéder en 3 temps :

1. insérer la colonne avec le nouveau nom avant la colonne qui va être supprimée

2. copier les données de l'ancienne colonne vers la nouvelle colonne
3. supprimer l'ancienne colonne

En SQL, cela donne ceci :

```
ALTER TABLE my_table ADD COLUMN new_name ... BEFORE old_name ;
UPDATE my_table SET new_name = old_name;
ALTER TABLE my_table DROP COLUMN old_name;
```

Il y a néanmoins un effet de bord avec le DROP COLUMN, car cette instruction déclenche un message système CPA32B2 nécessitant une intervention. Ce message d'interruption apparaît automatiquement quand on travaille en mode 5250 (via STRSQL), il est alors facile d'y répondre (en l'occurrence il faut répondre I pour Ignore car le message indique que des données vont être perdues, ce qui est en l'occurrence normal).

Par contre, il est impossible de répondre manuellement à ce type de message à partir d'un client SQL en mode graphique (comme System i Navigator ou SquirrelSQL). Heureusement, il existe une solution de contournement décrite ci-dessous :

- il faut tout d'abord qu'une réponse système automatique soit définie sur l'IBMi via la commande WRKRPYLE, pour le message CPA32B2
- ensuite, et avant l'exécution de l'ALTER TABLE, il faut exécuter la commande SQL suivante :

```
call qcmdexc ('CHGJOB INQMSGRPY(*SYSRPYL)', 26) ;
```

La documentation pour la valeur *SYSRPYL indique ceci :

« Le système vérifie, dans la liste des réponses système, si un poste existe pour tout message d'interrogation émis par ce travail. Si c'est le cas, il utilise la réponse de ce poste. Sinon, une réponse est obligatoire. »

On rappelle que le message renvoyé par l'ALTER TABLE.. DROP COLUMN.. est le CPA32B2. Si ce message a une réponse automatique définie sur l'IBM i, alors la commande CHGJOB permet d'en bénéficier au sein du travail relatif au code SQL en cours d'exécution.

6.7 Script de création de table

Pour illustrer les principales techniques étudiées dans ce chapitre, voici un exemple de script de création d'une table :

```
-- Script de création d'une table PRODUIT dans la bibliothèque MABIB

-- Création de la table avec une contrainte de clé primaire
CREATE TABLE MABIB/PRODUIT
( ID          DECIMAL(10 , 0)  NOT NULL WITH DEFAULT,
  LIBELLE     CHAR (30 )       NOT NULL WITH DEFAULT,
  PRIX        DECIMAL (13 , 2) NOT NULL WITH DEFAULT,
  UTI_MAJ     CHAR(18)         NOT NULL WITH DEFAULT USER,
  DAT_MAJ     DATE             NOT NULL WITH DEFAULT CURRENT DATE,
  HEU_MAJ     TIME             NOT NULL WITH DEFAULT CURRENT TIME,
  CONSTRAINT    PK_ID PRIMARY KEY( ID )
 ) ;

-- Ajout d'un commentaire sur la table
COMMENT ON TABLE MABIB/PRODUIT IS 'Fichier Produit' ;

-- Ajout d'une description sur la table
LABEL ON TABLE MABIB/PRODUIT IS 'Fichier Produit' ;

-- Ajout de commentaires sur les entête de colonnes de la table
COMMENT ON COLUMN MABIB/PRODUIT
( ID      IS 'Identifiant' ,
  LIBELLE IS 'Libellé' ,
  PRIX    IS 'Prix' ,
  UTI_MAJ IS 'Utilisateur MAJ' ,
  DAT_MAJ IS 'Date MAJ' ,
  HEU_MAJ IS 'Heure MAJ'
 ) ;

-- Ajout de labels sur les colonnes de la table
LABEL ON COLUMN MABIB/PRODUIT
( ID      IS 'Identifiant' ,
  LIBELLE IS 'Libellé' ,
  PRIX    IS 'Prix' ,
  UTI_MAJ IS 'Utilisateur MAJ' ,
  DAT_MAJ IS 'Date MAJ' ,
  HEU_MAJ IS 'Heure MAJ'
 ) ;

-- Ajout de descriptions sur les colonnes de la table
LABEL ON COLUMN MABIB/PRODUIT
( ID      TEXT IS 'Identifiant' ,
  LIBELLE TEXT IS 'Libellé' ,
  PRIX    TEXT IS 'Prix' ,
  UTI_MAJ TEXT IS 'Utilisateur MAJ' ,
  DAT_MAJ TEXT IS 'Date MAJ' ,
  HEU_MAJ TEXT IS 'Heure MAJ'
 ) ;
```

```
-- Ajout d'un index en clé unique sur la clé primaire de la table
CREATE UNIQUE INDEX MABIB/PRODUITL01
  ON  PRODUIT ( ID ASC ) ;

-- Ajout d'un index secondaire sur la table
CREATE INDEX MABIB/PRODUITL02
  ON  PRODUIT ( LIBELLE ASC, ID ASC ) ;
```

6.8 Alias

Un alias peut être considéré comme un raccourci vers un fichier.

Exemple :

On veut lister, par SQL tous les enregistrements du membre MBR2 du fichier FIC. Supposons que le fichier FIC contienne 2 membres (MBR1 et MBR2). MBR2 n'est pas le 1er membre de FIC, donc un simple « SELECT * FROM BIB/FIC » ne liste que les enregistrements du 1er membre de FIC (soit MBR1).

Pour pouvoir lister les enregistrements de MBR2, il faut soit utiliser OVRDBF avant de lancer l'instruction SQL, soit utiliser un alias SQL.

Pour créer un alias :

```
CREATE ALIAS MONALIAS FOR BIB.FIC (MBR2);
```

Il est possible de créer un alias temporaire, dont la durée de vie n'excèdera pas celle du traitement qui l'a créé :

```
CREATE ALIAS QTEMP.MONALIAS FOR BIB.FIC (MBR2);
```

La requête « **SELECT** * **FROM** QTEMP.MONALIAS » permet maintenant de lister les enregistrements du membre MBR2 de la table FIC.
En fait toutes les opérations SQL sur QTEMP/TOTO tel que UPDATE, DELETE, SELECT ...portent sur le membre MBR2 de BIB/FIC.

ATTENTION : DROP TABLE nom-alias détruit le fichier PHYSIQUE. Pour supprimer l'alias, il faut écrire DROP ALIAS nom-alias.

A noter : à partir de la V7R1, il devient possible de créer des alias pointant vers des tables se trouvant sur des systèmes distants :

```
CREATE ALIAS QTEMP.MONALIAS FOR ENVIR.BIB.FIC (MBR2);
```

6.9 Préparation d'un jeu de données avec Excel

Ce chapitre présente une astuce simple pour créer rapidement une table SQL et son contenu à partir d'une liste extraite d'une page HTML.
Il s'agit en l'occurrence de créer une table contenant une liste de pays.
Cette technique simple mais peu connue permet de constituer rapidement des jeux de données pouvant être utilisés dans le cadre de tests ou de petits projets.

On peut récupérer cette liste de pays sur Wikipedia, sous la rubrique ISO_3166-1 :

Mais pour gagner du temps, dans le cadre de cette formation, on pourra se référer au code SQL de la table LSTPAYS fourni en annexe du présent document. Ce source permettra de créer rapidement la table des pays, puis de l'exporter sous Excel (par exemple via le logiciel System i Navigator), pour pouvoir tester ce qui va suivre.

Après avoir copié-collé le tableau des pays dans Excel, et avoir éliminé les données qui ne nous intéressaient pas, on obtient le tableau suivant :

On peut bien sûr créer la table des pays sur DB2 for i en utilisant DDS, mais il est préférable de le faire plutôt en passant par SQL. Pour cela, démarrer une session SQL (via la commande STRSQL), puis exécutez les requêtes suivantes (remplacez « MABIB » par la bibliothèque que vous souhaitez utiliser pour vos tests) :

```
CREATE TABLE MABIB.LSTPAYS (
  CODFRA CHAR (3 ) NOT NULL WITH DEFAULT,
  CODISO CHAR (2 ) NOT NULL WITH DEFAULT,
  LIBELLE CHAR (50 ) NOT NULL WITH DEFAULT
)
```

Vous pouvez si vous le souhaitez ajouter un libellé à votre table LSTPAYS avec la requête

suivante :

```
COMMENT ON TABLE LSTPAYS IS 'TABLE DES PAYS' ;
```

Je vous propose également d'ajouter à votre table les indexs ci-dessous, qui vous permettront d'obtenir des requêtes SQL plus performantes :

```
CREATE INDEX MABIB.LSTPAYSL01 ON MABIB.LSTPAYS(CODFRA);
CREATE INDEX MABIB.LSTPAYSL02 ON MABIB.LSTPAYS(CODISO);
CREATE INDEX MABIB.LSTPAYSL03 ON MABIB.LSTPAYS(LIBELLE, CODFRA);
```

Vous pouvez exécuter ces requêtes de création sur DB2 Express C, en remplaçant par un « . », le « / » qui est spécifique à DB2 pour IBM i.

A titre d'information, le code SQL pour la création de la même table sous MySQL se présente de la façon suivante (remplacer « mabase » par la BD MySQL de votre choix) :

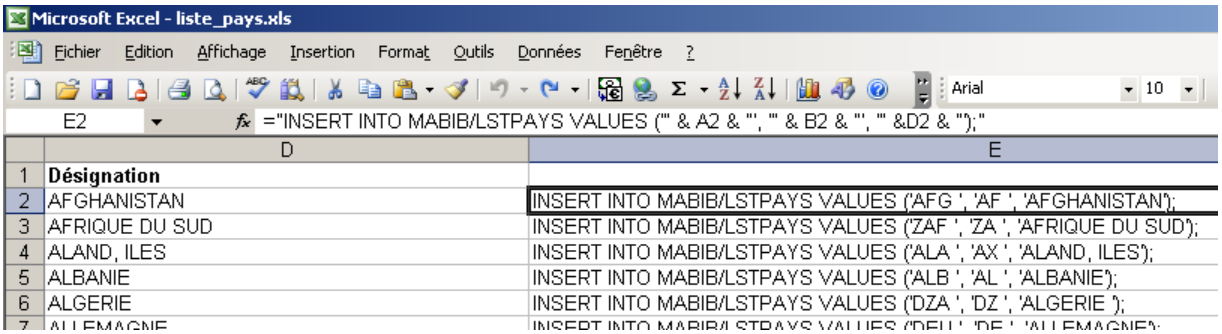
```
CREATE TABLE MABIB.LSTPAYS (
   codfra CHAR( 3 ) NOT NULL DEFAULT ' ',
   codiso CHAR( 2 ) NOT NULL DEFAULT ' ',
   libelle CHAR( 50 ) NOT NULL DEFAULT ' '
) ENGINE = MYISAM CHARACTER SET utf8 COMMENT = 'table des pays' ;

CREATE INDEX mabib.lstpaysl01 ON mabib.lstpays(codfra);
CREATE INDEX mabib.lstpaysl02 ON mabib.lstpays(codiso);
CREATE INDEX mabib.lstpaysl03 ON mabib.lstpays(libelle, codfra);
```

Revenons maintenant à notre fichier Excel pour le finaliser : nous souhaitons récupérer les colonnes A, B et D. Pour ce faire, nous allons créer une colonne supplémentaire dans laquelle nous allons utiliser une formule de concaténation, de manière à créer des requêtes INSERT pour chaque ligne du tableau Excel. Concrètement, la requête de concaténation se présente de la façon suivante :

```
="INSERT INTO MABIB/LSTPAYS VALUES ('" & A2 & "', '" & B2 & "', '" &D2 & "');"
```

Dès que la requête fonctionne pour une ligne, vous pouvez la dupliquer sur toutes les lignes en-dessous.
C'est sans doute plus parlant quand on le voit en action sur Excel :



Il faut ensuite copier le contenu de la colonne E dans le presse-papier, puis effectuer un « collage spécial » vers une colonne vierge d'Excel en utilisant l'option « par valeurs ». Vous obtenez ainsi un script SQL d'insertion de toutes les lignes du tableau Excel.

J'allais oublier un détail important : le problème des quotes, ou apostrophes. Je vous invite à regarder la colonne E se situant en face de la ligne de la « COTE D'IVOIRE ». Normalement, vous devriez avoir ceci :

```
INSERT INTO MABIB.LSTPAYS VALUES ('CIV ', 'CI ', 'COTE D'IVOIRE ');
```

Si vous y regardez de près, vous constaterez que vous avez un nombre d'apostrophes impair entre les parenthèses du mot-clé SQL VALUES. Donc cette requête SQL ne pourra pas fonctionner, pas plus sous DB2 que sous MySQL. Et on retrouve le même problème sur tous les libellés contenant une ou plusieurs apostrophes.

Vous pouvez corriger facilement le problème en utilisant la fonction Excel suivante :

```
=SUBSTITUE(D2;"'";"''")
```

ce qui nous donne la formule Excel suivante :

```
="INSERT INTO MABIB/LSTPAYS VALUES ('" & A2 & "', '" & B2 & "', '" &
SUBSTITUE(D2;"'";"''")
 & "');"
```

Après correction, la requête d'insertion pour la Côte d'Ivoire ressemblera à ceci :

```
INSERT INTO MABIB.LSTPAYS VALUES ('CIV ', 'CI ', 'COTE D''IVOIRE ');
```

La correction étant effectuée pour l'ensemble des lignes du tableau, vous pouvez recopier la colonne E dans le presse-papier Windows, puis effectuer un « collage spécial par valeur » dans une colonne vierge d'une autre feuille Excel.

Vous disposez maintenant d'un script SQL d'insertion prêt à l'emploi vous permettant d'alimenter la table SQL des pays, que vous pouvez sauvegarder au format texte. Vous pouvez ainsi le transférer sur votre serveur IBM i par tout moyen à votre convenance (par exemple FTP). Pensez à remplacer dans le script SQL « MABIB » par la bibliothèque de votre choix avant d'effectuer le transfert sur votre plateforme IBM i. Copiez le contenu du fichier dans un membre de fichier source (par exemple : MABIBSRC/QSQLSRC), dans le membre LSTPAYS, puis exécutez

le script via la commande OS/400 RUNSQLSTM, de la façon suivante :

```
RUNSQLSTM SRCFILE(MABIBSRC/QSQLSRC) SRCMBR(LSTPAYS) COMMIT(*NONE)
```

Vous risquez de rencontrer quelques problèmes dûs à des requêtes INSERT trop longues, que la commande RUNSQLSTM ne supporte pas. Certaines requêtes devront donc être légèrement modifiées de façon à être saisies sur 2 lignes, comme par exemple :

```
INSERT INTO MABIB.LSTPAYS VALUES ('COD ', 'CD ',
       'CONGO, LA REPUBLIQUE DEMOCRATIQUE DU ');
```

Si tout s'est bien passé, vous disposez donc d'une table DB2 contenant la liste des pays. Nous allons jouer un peu avec cette table au chapitre suivant.

6.10 Préparation d'un jeu de données avec SQL

Nous avons vu comment générer un jeu de données avec Excel, et si nous faisions maintenant la même chose avec SQL. Je vous propose donc de générer du SQL... à l'aide de SQL.

A partir de notre table DB2 des pays, nous pourrions générer assez facilement un script qui ressemblerait à ceci :

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AFG ', 'AF ', 'AFGHANISTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ZAF ', 'ZA ', 'AFRIQUE DU SUD');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ALA ', 'AX ', 'ALAND, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ALB ', 'AL ', 'ALBANIE');
...
```

Mais nous voulons faire un peu mieux que ça, en générant un script tel que celui ci-dessous :

```
INSERT INTO MYLIBRARY.LSTPAYS (CODFRA, CODISO, LIBELLE) VALUES
('AFG ', 'AF ', 'AFGHANISTAN'),
('ZAF ', 'ZA ', 'AFRIQUE DU SUD'),
('ALA ', 'AX ', 'ALAND, ILES'),
('ALB ', 'AL ', 'ALBANIE'),
...
```

Le code SQL ci-dessus offre 2 avantages par rapport à la version précédente :
- il est plus compact, et si vous voulez l'envoyer à quelqu'un par email, c'est un avantage non négligeable
- il est plus performant à l'exécution car le chemin d'accès déterminé par SQL pour l'insertion, n'est calculé qu'une seule fois et réutilisé pour toutes les lignes de données.

Pour générer la première ligne, la requête ci-dessous suffit :

```
select 'INSERT INTO MYLIBRARY.LSTPAYS (CODFRA, CODISO, LIBELLE) VALUES ' as DATA
from sysibm.sysdummy1
;
```

Cela nous donne l'occasion d'utiliser la table pivot SYSDUMMY1, qui est très utile dans des situations telles que celle-ci.

Pour les lignes suivantes, c'est un peu plus compliqué car il faut bien faire attention au nombre de guillemets :

```
select '(''' concat trim(codfra) concat ''','''' concat trim(codiso)
    concat ''','''' concat trim(replace(libelle, '''', ''''''))
    concat '''),' as data
from gjabase.LSTPAYS ;
```

Explication : La fonction trim() nous permet d'éliminer les blancs en début et surtout en fin de chaîne, l'ordre CONCAT utilisé de cette façon nous permet de concaténer différents éléments très facilement, et surtout il y a la fonction REPLACE() pour "doubler" les apostrophes sur les libellés tels que celui de la "Côte d'Ivoire".

Conseil : la mise au point de la requête peut être assez pénible du fait des apostrophes, aussi on recommandera de démarrer avec une version plus simple qui fonctionne, même si elle est incomplète, telle que celle ci-dessous :

```
select '(' concat trim(codfra) concat ',' concat trim(codiso)
   concat ',' concat trim(libelle) concat '),' as data
from gjabase.LSTPAYS ;
```

Ensuite seulement, doublez les apostrophes et insérez la fonction REPLACE() pour obtenir un code SQL valide.

Et pour obtenir la requête SQL INSERT complète en une seule passe, on peut coupler les 2 requêtes au moyen de la clause UNION :

```
select 'INSERT INTO MYLIBRARY.LSTPAYS (CODFRA, CODISO, LIBELLE) VALUES ' from
sysibm.sysdummy1
union
select '(''' concat trim(codfra) concat ''','''' concat trim(codiso) concat ''','''
concat trim(replace(libelle, '''', '''''')) concat '''),' as data
from gjabase.LSTPAYS ;
```

Ce qui nous donne à l'exécution le code SQL suivant :

```
INSERT INTO MYLIBRARY.LSTPAYS (CODFRA, CODISO, LIBELLE) VALUES
('AFG','AF','AFGHANISTAN'),
('ZAF','ZA','AFRIQUE DU SUD'),
('ALA','AX','ALAND, ILES'),
('ALB','AL','ALBANIE'),
('DZA','DZ','ALGERIE'),
('DEU','DE','ALLEMAGNE'),
...
```

On voit qu'il y a malgré tout un petit défaut, car la toute dernière ligne devrait se terminer par un point virgule, et non une virgule.
On peut bien sûr rectifier ce petit problème manuellement, mais pour le résoudre via SQL, il faut aborder différentes techniques, telles que les Common Table Expressions (CTE), techniques qui seront étudiées durant le cours SQL avancé. C'est pourquoi nous nous arrêterons là pour cet exemple (qui pourra être repris et finalisé durant le cours avancé, si vous le souhaitez).

7. Maintenance des données

I***NSERT INTO***

- avec choix de colonnes :

```
INSERT INTO tabempl (mat, nom, sal, sx)
  VALUES (80, 'PHILIPPE', 18500, 'M')
```

- sans choix de colonnes :

```
INSERT INTO tabempl
  VALUES (80, 'PHILIPPE', 'M', 990, 16000, '17/09/67')
```

ATTENTION : on préfèrera la première forme (avec choix de colonnes) à la seconde, car en cas de modification ultérieure dans l'ordre des colonnes, la requête INSERT continuera à fonctionner sans problème.

On peut aussi effectuer des insertions multiples, de la façon suivante :

```
INSERT INTO tabempl (mat, nom, sal, sx, date_nais)
  VALUES (10, 'PHILIPPE', 'M', 990, 16000, '17/09/67'),
         (20, 'SERGE', 'M', 990, 15000, '17/09/67'),
         (30, 'ROGER', 'M', 990, 10000, '17/09/67')
```

On peut ajouter par l'intermédiaire d'une vue (sauf restrictions indiquées au chapitre précédent) :

- la vue doit inclure toutes les colonnes NOT NULL
- les colonnes absentes comporteront la valeur indéfinie ou la valeur par défaut prévue

on peut indiquer explicitement :

- DEFAULT
- NULL
- USER (si colonne CHAR)
- CURRENT DATE/TIME/TIMESTAMP (si colonne D/T/TS)

on ne peut pas indiquer NULL ou DEFAULT pour une colonne NOT NULL

Alimentation par le résultat d'un SELECT

exemple 1 :

```
INSERT INTO empserv (noempl, nomempl, service)
```

```
   SELECT mat, nom, nomserv
     FROM tabempl, tabserv
     WHERE tabempl.srv = tabserv.srv
exemple 2 :
INSERT INTO TMP_001
 (SELECT A.EMPNO,A.LASTNAME FROM EMPLOYE A)
```

remarque : les littéraux numériques et les mots réservés (NULL, USER, DEFAULT...) sont à entrer tels quels, les littéraux alphanumériques ainsi que les littéraux D/T/TS sont à entrer entre apostrophes

7.1 INSERT

INSERT INTO

- avec choix de colonnes :

```
INSERT INTO tabempl (mat, nom, sal, sx)
  VALUES (80, 'PHILIPPE', 18500, 'M');
```

- sans choix de colonnes :

```
INSERT INTO tabempl
  VALUES (80, 'PHILIPPE', 'M', 990, 16000, '17/09/67');
```

ATTENTION : on préfèrera la première forme (avec choix de colonnes) à la seconde, car en cas de modification ultérieure dans l'ordre des colonnes, la requête INSERT continuera à fonctionner sans problème.

On peut aussi effectuer des insertions multiples, de la façon suivante :

```
INSERT INTO tabempl (mat, nom, sal, sx, date_nais)
  VALUES (10, 'PHILIPPE', 'M', 990, 16000, '17/09/67'),
         (20, 'SERGE', 'M', 990, 15000, '17/09/67'),
         (30, 'ROGER', 'M', 990, 10000, '17/09/67');
```

On peut ajouter des données à une table par l'intermédiaire d'une vue, sous certaines conditions, telle que :

- la vue doit inclure toutes les colonnes NOT NULL
- les colonnes absentes comporteront la valeur indéfinie ou la valeur par défaut prévue

on peut indiquer explicitement :

- DEFAULT
- NULL
- USER (si colonne CHAR)
- CURRENT DATE/TIME/TIMESTAMP (si colonne D/T/TS)

on ne peut pas indiquer NULL ou DEFAULT pour une colonne NOT NULL

Alimentation par le résultat d'un SELECT

Exemple 1 :

```
INSERT INTO empserv (noempl, nomempl, service)
   SELECT mat, nom, nomserv
       FROM tabempl, tabserv
       WHERE tabempl.srv = tabserv.srv;
```

Exemple 2 :

```
INSERT INTO TMP_001
 (SELECT A.EMPNO,A.LASTNAME FROM EMPLOYE A);
```

- remarque : les littéraux numériques et les mots réservés (NULL, USER, DEFAULT...) sont à entrer tels quels, les littéraux alphanumériques ainsi que les littéraux D/T/TS sont à entrer entre apostrophes

7.2 UPDATE

UPDATE

- mise à jour d'une colonne dans une ligne :

```
UPDATE tabempl SET srv = 977 WHERE mat = 20;
```

- mise à jour de plusieurs colonnes dans une ligne :

```
UPDATE tabempl SET srv = 911, sal = 13500 WHERE mat = 30;
```

- mise à jour d'une colonne dans plusieurs lignes :

```
UPDATE tabempl SET sal = sal * 1,1 WHERE sx = 'F';
```

- on peut ajouter par l'intermédiaire d'une vue (sauf restrictions indiquées au chapitre précédent) :
  - la vue doit inclure toutes les colonnes NOT NULL
  - les colonnes absentes comporteront la valeur indéfinie ou la valeur par défaut prévue

- on peut indiquer explicitement :
  - DEFAULT
  - NULL
  - USER (si colonne CHAR)
  - CURRENT DATE/TIME/TIMESTAMP (si colonne D/T/TS)

- mise à jour à partir d'un SELECT

  - mise à jour d'une colonne dans une ligne :

```
UPDATE tabempl
   SET srv = (SELECT srv FROM tabserv WHERE nomsrv = 'VENTES')
   WHERE mat = 60 ;
```

  - mise à jour de plusieurs colonnes dans toutes les lignes :

```
UPDATE dept d
    SET (maxheur, totheur) =
       (SELECT MAX(empheur), SUM(empheur)
         FROM relevheure WHERE d.mat = mat
       --GROUP BY mat);
```

Dans l'exemple ci-dessus, le GROUP BY est accessoire et peut carrément être occulté car le filtre dans la clause WHERE fait office de jointure entre les tables dept et relevheure. (cf. exemple plus détaillé en fin de chapitre)

- utilisation de l'expression CASE :

```
UPDATE tabempl
     SET sal = CASE
       WHEN sx = 'F' THEN sal * 1,05
       WHEN sx = 'M' THEN sal * 1,08
       ELSE sal * 2
       END;
```

Exemple détaillé reprenant la mise à jour de la table "dept" à partir de données de la table "relevheure" :

```
create table formation/dept (
mat integer,
maxheur decimal(10, 0),
totheur decimal(10, 0)
) ;

create table formation/relevheure (
mat integer,
empheur decimal (10, 0)
);

insert into formation/dept (mat, maxheur, totheur) values
(1, 0, 0),
(2, 0, 0),
(3, 0, 0);

insert into formation/relevheure (mat, empheur) values
(1, 20),
(2, 30),
(2, 31),
(3, 52),
(3, 50),
(4, 100)
;

UPDATE formation/dept d
SET (maxheur, totheur) =
       (SELECT MAX(x.empheur), SUM(x.empheur)
              FROM formation/relevheure x WHERE d.mat = x.mat
             -- GROUP BY x.mat
);
```

Dans le cas où on souhaite mettre à jour une colonne de la table dept à partir d'une colonne de la table relevheure, on peut utiliser la clause FETCH FIRST 1 ROW ONLY dans la sous-requête pour ne récupérer qu'une occurrence par matricule :

```
-- requête ne peut fonctionner car la sous-requête renvoie plus d'une ligne pour
chaque matricule
update formation/dept a set a.maxheur =
  ( select b.empheur from formation/relevheure  b where a.mat = b.mat order by 1 )
;

-- solution de contournement consistant à récupérer la 1ère ligne de relevé d'heure
(incluant un ORDER BY) pour chaque matricule
update formation/dept a set a.maxheur =
  ( select b.empheur from formation/relevheure  b where a.mat = b.mat order by 1 desc
fetch first 1 row only)
;
```

7.3 DELETE

DELETE

- suppression d'une ligne :

```
DELETE FROM tabempl WHERE mat = 50;
```

- suppression de plusieurs lignes :

```
DELETE FROM tabempl WHERE nom LIKE 'R%';
```

- suppression de toutes les lignes (équivalent à la commande AS/400 CLRPFM) :

```
DELETE FROM tabempl;
```

- suppression d'une ligne par sous-sélection :

```
DELETE FROM tabserv WHERE srv =
  (SELECT srv FROM tabempl WHERE mat = 50) ;
```

- suppression de plusieurs lignes par sous-sélection :

```
DELETE FROM tabempl WHERE srv =
  (SELECT srv FROM tabserv WHERE nomsrv = 'VENTES');
```

A partir de la V5R3, et dans le cas d'un DELETE sans WHERE, le système pourra effectuer l'équivalent d'un CLRPFM, si certaines conditions sont remplies :

- la table citée n'est pas une vue
- il existe un grand nombre de lignes
- pas de curseur ouvert sur le fichier par le même travail
- pas de verrouillage par un autre travail
- pas de déclencheur en DELETE
- la table n'est pas parente avec un ON DELETE CASCADE, SET NULL ou SET DEFAULT
- le profil du travail dispoe d'*OBJMGT ou *OBJALTER sur la table concernée

Conséquences :

- le DELETE est beaucoup plus rapide
- pas de postes individuels DL dans le journal (mais un poste CR, ou CG en contrôle de validation)
- COMMIT/ROLLBACKsupporté (mais pas d'APY/RMVJRNCHG possibles)

IMPORTANT : IBM annonce l'arrivée de l'ordre TRUNCATE sur la V7R2, qui équivaudra strictement à un CLRPFM, sans les restrictions indiquées ci-dessus pour le DELETE.

Autres exemples de suppression pris sur le site foothing.net :

Supprimer tous les enregistrements de la table t1 ayant une correspondance dans la table t2, C1 étant la zone identifiant la correspondance.

```
DELETE FROM T1 A
WHERE EXISTS
(
  SELECT * FROM T2 B
  WHERE B.C1 = A.C1
) ;
```

Supprimer tous les enregistrements de la table T1 sans correspondance dans la table T2, C1 étant la zone identifiant la correspondance.

```
DELETE FROM T1 A
WHERE NOT EXISTS
(
  SELECT * FROM T2 B
  WHERE B.C1 = A.C1
);
```

Supprimer les doublons de la table T1 sur la valeur de la colonne COLX

```
DELETE FROM T1 A
WHERE RRN(A) NOT IN
( SELECT MAX( RRN(B) ) FROM T1 B WHERE A.COLX = B.COLX );
```

7.4 MERGE

Apparu sur DB2 for i à partir de la V7R1, l'ordre SQL MERGE permet de combiner plusieurs instructions INSERT, DELETE et UPDATE au sein d'une seule instruction MERGE.

Exemple de requête MERGE :

```
MERGE INTO My_LIBRARY.testmerge2 a
 USING (SELECT macle , codea , coden FROM My_LIBRARY.testmerge ) b
 ON a.macle = b.macle
 WHEN MATCHED and a.codea = 'A1'  THEN
    DELETE
 WHEN MATCHED and a.codea = 'A2'  THEN
 UPDATE SET
  a.codea = 'X2' ,
  a.coden = 9999
 WHEN MATCHED and a.codea <> 'A1'  and a.codea <> 'A2'    THEN
 UPDATE SET
  a.codea = b.codea ,
  a.coden = a.coden + b.coden
WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden )
 VALUES ( b.macle , b.codea , b.coden )
;
```

Le principe de fonctionnement de MERGE sera étudié plus en détail dans le cours "SQL Avancé".

8. Rappels sur les fonctions SQL

DB2 fournit en standard un certain nombre de fonctions.
Parmi les fonctions proposées, on trouve :

- des fonctions d'agrégation qui s'appliquent aux colonnes, comme par exemple la fonction AVG (moyenne) ,
- des fonctions « opérateurs », comme par exemple le « + » ,
- des fonctions de conversion (en anglais : « casting functions ») comme par exemple la fonction DECIMAL ,
- des fonctions de manipulation de chaînes, comme par exemple la fonctions SUBSTR ,
- etc…

8.1 Fonctions scalaires

Les fonctions scalaires (CONCAT, CAST, DATE, etc...) :

FONCTION	DESCRIPTION
ABS	Retourne la valeur absolue de l'expression passée en paramètre
CAST	Conversion d'un type vers un autre
CEIL/CEILING	Entier supérieur ou égal à l'expression passée en paramètre
CHAR	Conversion vers caractère
COALESCE	Remplacement de valeur indéfinie (équivalent à VALUE, équivalent également à IFNULL mais sans la limite de 2 paramètres)
CONCAT	Concaténation
DATE	Conversion d'un type en date
DECIMAL	Reformatage numérique
DIFFERENCE	renvoi de 0 à 4 selon le degré de proximité du son de 2 expressions (4 si les sons sont quasi identiques)
DIGITS	Conversion numérique -> alpha (important : DIGITS permet de conserver les zéros à gauche, contrairement à la fonction CHAR)
GRAPHIC	(expression alpha, [longueur], [CCSID]) : transforme une chaîne en DBCS ou UCS-2 (UNicode)
HEX	Retourne le contenu d'une colonne en hexadécimal
IFNULL	Remplacement de valeur indéfinie (équivalent aux fonctions VALUE et COALESCE mais n'accepte que 2 paramètres contrairement aux 2 autres fonctions)
INT/INTEGER/FLOOR	Entier inférieur ou égal
LEFT	Partie gauche d'une chaine, exemple : LEFT(colonne, 5)
LENGTH	Retourne la longueur d'une colonne
LOCATE	Renvoi de la position d'une sous-chaîne
LOWER	Transformation en minuscules

LTRIM	Suppression des espaces à gauche
MAX et MIN	Maximum et minimum d'une suite
PI()	fournit la valeur de PI en format FLOAT
RAND	renvoie un nombre aléatoire entre 0 et 1 de type FLOAT
RIGHT	Partie droite d'une chaîne, exemple : RIGHT(colonne, 5)
ROUND	Arrondi commercial
RRN	Renvoi du rang de la ligne (n° relatif d'enregistrement)
RTRIM	Suppression des espaces à droite

FONCTION (suite)	DESCRIPTION
SIGN	Renvoie -1, 0 ou 1 selon que l'expression est négative, nulle ou positive
SOUNDEX	Codage d'une chaîne de caractères
SPACE	renvoie une chaîne de type VARCHAR contenant le nombre d'espaces indiqué (exemple : SPACE(10))
STRIP/TRIM	Suppression de caractères
SUBSTR	Sous-chaîne de caractères
TRANSLATE	Remplacement de caractères
TRIM	Suppression des blancs à gauche et à droite
UPPER	Transformation en majuscules
VALUE	Remplacement de valeur indéfinie (équivalent à COALESCE, équivalent également à IFNULL mais sans la limite de 2 paramètres)

Quelques exemples d'utilisation des fonctions :

CHAR permet de transformer une valeur numérique en alphanumérique.
Elle autorise l'usage d'un second paramètre, permettant de préciser le format du séparateur décimal. Par exemple, pour remplacer la virgule par un point dans le salaire des employés :

```
SELECT CHAR(sal, '.') FROM tabempl
```

DECIMAL permet de reformater une colonne numérique :

```
SELECT DECIMAL(sal, 7, 2) AS salmoyen FROM tabempl
```

On peut imbriquer plusieurs fonctions :

```
SELECT INT(SUM(sal)) AS totsal FROM tabempl
SELECT DECIMAL(AVG(sal), 7, 2) AS salmoyen FROM tabempl
```

MIN et MAX acceptent un second argument, qui fixe la limite maximale de recherche pour MIN, et la limite minimale de recherche pour MAX :

```
SELECT MIN(srv, 500), MAX(srv, 920) FROM tabempl
```

LEFT(nom, x) est équivalent à SUBSTR(nom, 1, x)
RIGHT(nom, x) est équivalent à SUBSTR(nom, LENGTH(nom)-x+1, x)

STRIP

Supprime tout espace (ou autre caractère spécifié) au début ou à la fin d'une expression.
1er argument : nom de colonne ou expression
2ème argument (optionnel) : code suppression

BOTH ou B	au début et à la fin (défaut)
LEADING ou L	au début
TRAILING ou T	à la fin

3ème argument (optionnel) :
caractère à supprimer (espace par défaut)
Si col contient « xxLIBELLEx »
STRIP(col, B, 'x') fournit « LIBELLE »
Remarque :

- TRIM(B 'x' FROM col) fournirait le même résultat
- il existe aussi LTRIM et RTRIM

LENGTH

Fournit la longueur d'une colonne ou d'une expression.

- Si la colonne est de longueur fixe, LENGTH(col) indique la longueur de cette colonne.
- Si la colonne est de longueur variable, LENGTH(col) indique la longueur du contenu de cette colonne.
- Pour obtenir la longueur du contenu d'une colonne de longueur fixe : LENGTH(RTRIM(col))
- Pour calculer la longueur moyenne des contenus d'une colonne de longueur fixe : AVG(LENGTH(RTRIM(col)))
- Pour calculer la longueur moyenne d'une colonne de longueur variable : AVG(LENGTH(col))

```
SELECT length(trim(CLNOM)), length(rtrim(CLNOM)),
       length(ltrim(CLNOM)), length(CLNOM), CLNOM FROM clip
```

LENGTH	LENGTH	LENGTH	LENGTH(CLNOM)	nom
4	4	24	24	MAKI
6	6	24	24	SEVEAU
14	14	24	24	PALFRAY MURIEL
13	13	24	24	RAOUL PATRICK
17	17	24	24	MOUSSET CATHERINE
12	12	24	24	BAAS ESNAULT

LOWER et UPPER

UPPER (ou UCASE) transcode une chaîne de caractères en minuscules.

```
SELECT mat, LOWER(nom) FROM tabempl WHERE mat = 50
```

LOWER (ou LCASE) transcode une chaîne de caractères en minuscules.

LOCATE

Fournit la position d'une sous-chaîne :

```
SELECT nom, LOCATE('A', nom, 1) as pos_a FROM tabempl ORDER BY mat
```

Par exemple, pour extraire le nom d'un contact se trouvant à gauche du « @ » d'une adresse email, ainsi que le domaine se trouvant à droite du « @ » :

```
SELECT adrmail,
       substring(adrmail, 1, locate('@', adrmail) -1 ) as nom,
       substring(adrmail, locate('@', adrmail) +1,
          length(adrmail)) as domaine
 FROM contacts
```

ADRMAIL	NOM	DOMAINE
Alan.K.Buccannan@rbs.co.uk	Alan.K.Buccannan	rbs.co.uk
i.rankin@napier.ac.uk	i.rankin	napier.ac.uk
P.Bhardwaj@napier.ac.uk	P.Bhardwaj	napier.ac.uk
Scott.Kemmer@napier.ac.uk	Scott.Kemmer	napier.ac.uk

Remarque : POSSTR(nom, 'A') est équivalent à POSITION('A' IN nom), et également équivalent à LOCATE ('A', nom, 1)

La requête ci-dessus peut donc s'écrire :

```
SELECT adrmail,
       substring(adrmail, 1, position('@' in adrmail) -1 ) as nom,
       substring(adrmail, position('@' in adrmail) +1,
          length(adrmail)) as domaine
 FROM contacts
```

TRANSLATE

Substitue un ou plusieurs caractères.

Exemple : remplacer dans « nom » les « A » par des « y » et les « R » par des « z ».

```
SELECT mat, TRANSLATE(nom, 'yz', 'AR') FROM tabempl
```

RRN

Renvoie le rang de la ligne dans la table.

```
SELECT mat, nom, RRN(tabempl) FROM tabempl ORDER BY nom
```

Récupérer la dernière ligne de la table :

```
SELECT * FROM A WHERE RRN(A) IN (SELECT MAX(RRN(A)) FROM A)
```

IF NULL

Renvoie la première valeur non indéfinie de 2 arguments.
Dans l'exemple ci-dessous, si une ligne a une valeur indéfine pour SRV, alors la valeur « 0 » est affichée :

```
SELECT nom, IFNULL(srv, 0) FROM tabempl WHERE sx = 'M'
```

Remarque : la fonction IFNULL est sensiblement équivalente aux fonctions COALESCE et VALUE mais IFNULL n'accepte que 2 arguments alors que COALESCE et VALUE en acceptent beaucoup plus.

ROUND

La fonction ROUND permet de supprimer les chiffres significatifs d'un nombre en précisant le nombre de décimales à conserver. Si le paramètre d'arrondi est positif, il indique le nombre de chiffres à conserver à droite de la virgule. S'il est négatif, le nombre est arrondi à la puissance de 10 correspondante. Exemple :

```
SELECT round(35,150, 1) as round_pos, round(35, -1) as round_neg
 FROM qsqptabl
```

ROUND_POS	ROUND_NEG
35,200	40

N.B. : la technique de l'arrondi négatif peut être utilisée pour agréger des données selon des catégories qui n'existent pas dans la table à traiter. Par exemple pour réduire des âges à la tranche inférieure à laquelle ils appartiennent, on peut procéder comme suit :

```
CREATE TABLE TSTROUND (AGE INTEGER NOT NULL WITH DEFAULT)
INSERT INTO  TSTROUND (AGE) VALUES(35), (34), (41), (55)

SELECT AGE , round(AGE, -1) FROM TSTROUND
```

AGE	ROUND
35	40
34	30
41	40
55	60

```
SELECT trim(char(classe-5)) concat '-' concat trim(char(classe+4))
       as tranche
 FROM ( SELECT round(age, -1) AS classe FROM TSTROUND ) t
```

TRANCHE
35-44
25-34
35-44
55-64

SOUNDEX et DIFFERENCE

SOUNDEX permet de coder le début d'une chaîne de caractères sur 4 octets.

DIFFERENCE(arg1, arg2) permet de connaître la proximité des SOUNDEX respectifs de 2 arguments, de 0 (minimum), à 4 (maximum).

```
SELECT nom, soundex(nom), soundex('ANDRE'), difference(nom, 'ANDRE')
 FROM tabempl
```

NOM	SOUNDEX(NOM)	SOUNDEX('ANDRE')	DIFFERENCE
ANNIE	A500	A536	2
CLAUDE	C430	A536	1
DANIELE	D540	A536	1
JACQUES	J220	A536	0
MARC	M620	A536	0

CONCATENATION

CONCAT ou le symbole !!

Permet l'assemblage de colonnes, constantes, valeurs calculées.

```
SELECT RTRIM(nom) CONCAT '-' CONCAT DIGITS(DEC(srv, 3, 0))
  FROM tabempl
  WHERE srv IS NOT NULL
```

Remarques :

- le symbole !! est spécifique à la plateforme IBM i et permet de pallier le fait que les doubles barres verticales (||) ne sont pas supportées en mode natif IBM i. Pour une meilleure portabilité des requêtes entre les différentes plateformes, on recommandera plutôt l'usage de la fonction CONCAT qui fonctionne dans tous les cas.

- la fonction CONCAT peut aussi être utilisée avec la syntaxe suivante, qui n'autorise que 2 paramètres : `CONCAT(arg1, arg2)`

- la fonction CONCAT peut être utilisée également dans la clause WHERE. Dans l'exemple ci-

dessous, la constante « Alan » est recherchée au début de « adrmail ». Mais cette technique peut tout aussi bien être utilisée sur une variable (c'est d'ailleurs le principal intérêt) :

```
SELECT adrmail
 FROM  contacts
 WHERE adrmail LIKE CONCAT('Alan', '%')
```

est équivalent à :

```
SELECT adrmail
 FROM  contacts
 WHERE adrmail LIKE 'Alan' CONCAT '%'
```

CASE

N.B. : voir également le chapitre qui s'intitule « conversion et comparaison de date » pour d'autres exemples d'utilisation de la clause CASE… WHEN..

CASE sert à effectuer des traitements conditionnés.

```
SELECT nom, sal, CASE
   WHEN   sal <=  14000  THEN  decimal(sal * 0,1, 7, 2)
   WHEN   sal <=  15000  THEN  decimal(sal * 0,05, 7, 2)
   ELSE  0
   END  AS  rallonge
 FROM tabempl
```

CASE peut aussi servir à expliciter des codes :

```
SELECT mat, nom, CASE srv
   WHEN  901  THEN 'Compta'
   WHEN  911  THEN 'Ventes'
   WHEN  977  THEN 'Manuf'
   ELSE  'Autres'
   END  AS  service
 FROM  tabempl
 ORDER BY nom
```

```
MAT   NOM         SERVICE
10    ANNIE Ventes
60    CLAUDE      Autres
40    DANIELE     Manuf
...
```

8.2 Fonctions d'agrégation

Format : Fonction(expression)

SUM	Totalisation
AVG	Valeur moyenne
MIN	Valeur minimum
MAX	Valeur maximum
COUNT	Comptage (15,0)
COUNT_BIG	Comptage (31,0)
STDDEV	Ecart-type
VAR	Variance

MIN, MAX, COUNT et COUNT_BIG peuvent s'appliquer à des colonnes alphanumériques.

Exemples :

```
SELECT MIN(sal), MAX(sal), SUM(sal) FROM tabempl

SELECT COUNT(*) FROM tabempl

SELECT COUNT(sx), COUNT(DISTINCT sx) FROM tabempl
```

Exemple de fonction couplée avec un GROUP BY

```
SELECT sx, AVG(sal) FROM tabempl GROUP BY sx
```

IMPORTANT : les colonnes citées dans SELECT doivent être des fonctions de colonnes ou des colonnes du GROUP BY.

Si un classement est nécessaire, indiquer ORDER BY :

```
SELECT sx, AVG(sal) FROM tabempl GROUP BY sx ORDER BY AVG(sal)
```

La clause HAVING

Groupage sous condition : la clause HAVING implique toujours l'utilisation de GROUP BY

```
SELECT sx, AVG(sal)
 FROM tabempl
 GROUP BY sx
 HAVING COUNT(*) > 2


SELECT sx, AVG(sal)
 FROM tabempl
 GROUP BY sx
 HAVING AVG(sal) >= 14500
```

9. Manipulation des dates

9.1 Comparaison avec la date courante

Comparer des champs numériques et caractères avec la date courante

```
CREATE TABLE Testdate (
  PKCol    Int    Primary Key,
  Decdate  Decimal( 9, 0),
  Chardate Char(8) ) ;
INSERT INTO Testdate Values(1, 20090711, '20090711') ;
INSERT INTO Testdate Values(2, 20090712, '20090712') ;
INSERT INTO Testdate Values(3, 20090713, '20090713') ;

SELECT * FROM TESTDATE ;
```

PKCOL	DECDATE	CHARDATE
1	20.090.711	20090711
2	20.090.712	20090712
3	20.090.713	20090713

Instruction SQL pour comparer la date système par rapport à un champ numérique (DECDATE) :

```
SELECT * FROM Testdate
  WHERE Decdate = integer(replace(char(curdate(), ISO), '-', ''));
```

N.B. : à partir de la V5R3, on peut supprimer la function integer(). Cette dernière devient inutile car DB2 fait un CAST implicite avant comparaison des 2 champs, ce qui donne :

```
SELECT * FROM Testdate
  WHERE Decdate = replace(char(curdate(), ISO), '-', '');
```

Pour la comparaison de la date courante avec la colonne CHARDATE, le principe est similaire, mais plus simple, puisqu'on n'a pas besoin de passer par une conversion numérique (implicite ou non) :

```
SELECT * FROM Testdate
  WHERE Chardate = replace(char(curdate(), ISO), '-', '');
```

9.2 Conversion de dates

Conversion d'une date en format alphabétique :

Grâce la fonction CHAR, on peut formater de différentes manières des données de type date, heure et étiquettes temporelles (timestamp) selon la valeur du second paramètre :

```
SELECT char(current date, local) as fmt_local,
       char(current date, iso ) as fmt_iso,
       char(current date, usa) as fmt_usa,
       char(current date, jis) as fmt_jis,
       char(current date, eur) as fmt_eur
FROM sysibm/sysdummy1 ;
```

Le résultat de la requête ci-dessus est le suivant :

```
FMT_LOCAL  FMT_ISO     FMT_USA     FMT_JIS     FMT_EUR
 10/12/08  2008-12-10  12/10/2008  2008-12-10  10.12.2008
```

Conversion d'une date au format JJ/MM/AAAA :

(pour obtenir une date avec le mois sur 2 chiffres, même si sa valeur est inférieure à 10)

```
SELECT DAT_FIN, (DAY(A.DAT_FIN) CONCAT '/' CONCAT
      SUBSTRING(CHAR(A.DAT_FIN), 6, 2)  CONCAT '/' CONCAT
      YEAR(A.DAT_FIN) ) AS DAT_FIN_2
FROM ...
```

Conversion d'une date en format alphabétique sans formatage :

```
SELECT
  char(year(current date) * 10000 + month(current date) * 100
   + day(current date)) as date1,
  year(current date) CONCAT month(current date) CONCAT day(current date) as date2
FROM sysibm/sysdummy1 ;
```

```
DATE1        DATE2
20081210     20081210
```

Conversion d'une date en format numérique standard (donc non formaté) :

```
SELECT
  year(current date) * 10000 + month(current date) * 100
   + day(current date) as date1,
  dec( year(current date) CONCAT month(current date) CONCAT day(current date), 8, 0)
    as date2
FROM sysibm/sysdummy1 ;
```

```
     DATE1          DATE2
20.081.210     20.081.210
```

N.B. : dans les 2 techniques de conversion présentées ci-dessus, DATE1 est obtenue par calcul, et DATE2 par concaténation. Ce principe, appliqué ici à la date courante, peut aussi être appliqué à des champs dates stockés séparément dans une table DB2. Difficile d'avancer une solution plutôt que l'autre en termes de performances, on peut éventuellement préconiser d'utiliser la technique du calcul si les colonnes initiales sont déjà de type numérique, et d'utiliser la concaténation si les colonnes initiales sont de type alphanumérique.

Avec DB2 LUW, il est également possible d'utiliser la fonction To_Date() qui offre l'avantage de définir le format de la date dans la chaine transmise en second paramètre.
Exemples :

```
select TO_DATE('15-02-2014', 'DD-MM-YYYY') from sysibm.sysdummy1; -- 2006-02-15
00:00:00.0

select TO_DATE('20140624', 'YYYYMMDD') from sysibm.sysdummy1; -- 2014-06-24
00:00:00.0
```

9.3 Différences entre fonctions DAYS et DATE

La fonction DATE permet de convertir un entier en DATE. Les entiers autorisés vont de 1 à 3.652.059, où 1 représente le premier janvier de l'an 1. La fonction DAYS effectue la conversion inverse :

```
SELECT date(716194), days('1961-11-15') FROM sysibm.sysdummy1 ;
```

Le résultat de la requête ci-dessus est le suivant :

```
DATE ( 716194 )           DAYS
   15/11/61             716.194
```

ATTENTION : les fonctions DAYS et DATE permettent toutes deux de calculer l'écart entre 2 dates, mais DATE renvoie un résultat exprimé en nombre d'années, de mois et de jours. La fonction DAYS renvoie elle strictement le nombre de jours entre les 2 dates :

```
SELECT  date(current date) - date('2013-01-01') as calcul1,
        days(current date) - days('2013-01-01') as calcul2
FROM sysibm.sysdummy1 ;

  CALCUL1          CALCUL2
   10.523              539
```

Résultats obtenus pour CURRENT DATE égal au 24/06/2014.

```
La valeur 10523 obtenue sur CALCUL1 doit se lire ainsi :
   -  1 année
   -  5 mois
   -  23 jours
```

9.4 Conversion de numérique vers date

Une date stockée dans un champ numérique (de type 8.0) doit faire l'objet d'une conversion dans un format compatible avec la norme ANSI/ISO, si on souhaite par exemple lui appliquer des fonctions SQL de calcul de date.

Par exemple, si l'on considère une zone DATEXP, de type numérique 8.0, contenant des dates au format SSAAMMJJ. On va séparer l'année, le mois, et le jour avec la fonction SUBSTR, puis les reconcaténer avec des tirets intermédiaires, ce qui permet d'obtenir une date proche du format ANSI/ISO.

```
SELECT (substr(DATEXP, 1, 4) concat '-' concat
        substr(datexp, 5, 2) concat '-' concat
        substr(datexp, 7, 2)) as newdate
 FROM matable ;
```

Le résultat est le suivant (erroné quand la date d'origine est à zéro) :

```
NEWDATE
2007-08-03
2000-05-31
2000-05-31
2000-05-31
0   -  -
2000-06-06
…
```

On peut dès lors appliquer l'ordre CASE pour ne réaliser la conversion que dans le cas où la date est différente de zéro :

```
SELECT datexp,
       case when datexp = 0
          then null
          else date(substr(datexp, 1, 4) concat '-' concat
                    substr(datexp, 5, 2) concat '-' concat
                    substr(datexp, 7, 2)) end as newdate
FROM matable ;
```

Le résultat est le suivant (si DATEXP = 0 alors NEWDATE = null) :

```
 DATEXP    NEWDATE
 20070803  03/08/07
 20000531  31/05/00
 20000531  31/05/00
 20000531  31/05/00
```

```
         0    -
 20000606   06/06/00
```

Pour la conversion au format hh:mm:ss d'une heure stockée dans un format numérique (6.0), on utilisera une requête du type :

```
SELECT heunum,
       case when heunum = 0
          then null
          else char(substr(heunum, 1, 2) concat ':' concat
                    substr(heunum, 3, 2) concat ':' concat
                    substr(heunum, 5, 2)) end as newheure
FROM testdates ;
```

9.5 Conversion d'alpha vers date

Les fonctions DATE, TIME et TIMESTAMP attendent des paramètres de type donnée temporelle (date, heure, et étiquette temporelle). Elle n'acceptent que des chaînes qui sont converties implicitement en valeurs de ces types.

```
SELECT date('2008-01-14'), time('15:44:11'),
       timestamp('2008-01-14-15.44.11.985064')
FROM sysibm/sysdummy1 ;
```

Le résultat de la requête ci-dessus est le suivant :

```
DATE      TIME ( '15:44:11' )  TIMESTAMP
14/01/08       15:44:11        2008-01-14-15.44.11.985064
```

Les champs Date retournés par des commandes OS/400 telles que DSPFD ou DSPOBJD sont de type alphanumérique, de longueur 6, et de format MMDDYY (Month – Day – Year). Ces champs nécessitent de mettre en œuvre une mécanique de conversion similaire à celle présentée au chapitre précédent. Par exemple, pour convertir en champ numérique la colonne alphanumérique ODCDAT (date de création) obtenue par la commande DSPOBJD, on peut utiliser la requête suivante :

```
SELECT ODCDAT , case
  when substr(odcdat , 5, 2) = '  ' then 0
  when substr(odcdat , 5, 2) <= '50' then decimal
    ( '20' concat substr(odcdat , 5, 2) concat
      substr(odcdat, 1, 2) concat substr(odcdat, 3, 2))
  when substr(odcdat , 5, 2) > '50' then decimal
    ( '19' concat substr(odcdat , 5, 2) concat
      substr(odcdat, 1, 2) concat substr(odcdat, 3, 2))
```

```
  else 0
  end as DATECNV
 FROM bib/dspobjdout ;
```

Creation Date	DATECNV
101507	20071015
101507	20071015
110907	20071109
121707	20071217
110907	20071109
120403	20031204
	0
121499	19991214

Pour convertir le même champ ODCDAT en véritable champ de type Date, on pourra utiliser la requête suivante :

```
SELECT ODCDAT , case
  when substr(odcdat , 5, 2) = '  ' then null
  when substr(odcdat , 5, 2) <= '50' then date
    ( '20' !! substr(odcdat , 5, 2) !! '-' !!
      substr(odcdat, 1, 2) !! '-' !! substr(odcdat, 3, 2))
  when substr(odcdat , 5, 2) > '50' then date
    ( '19' !! substr(odcdat , 5, 2) !! '-' !!
      substr(odcdat, 1, 2) !! '-' !! substr(odcdat, 3, 2))
  else null
  end as DATECNV
FROM bib/dspobjdout;
```

```
Creation  DATECNV
Date
 101507   15/10/07
 101507   15/10/07
 110907   09/11/07
 121707   17/12/07
 110907   09/11/07
 120403   04/12/03
            Null
 121499   14/12/99
```

Technique de conversion spécifique à Excel :

Les fichiers Excel extraits de l'IBM i via Client Access contiennent généralement des dates dans un format peu lisible, et peu exploitable car de type texte, tel que 20.080.305.

Formule pour convertir cette date au format numérique 05032008 :

=STXT(X1;9;2)*1000000+((STXT(X1;6;1)*10+STXT(X1;8;1))*10000)+(STXT(X1;1;2)*100+STXT(X1;4;2))

Formule pour convertir cette date au format numérique 20080305 :

=(STXT(X1;1;2)*100+STXT(X1;4;2))*10000+((STXT(X1;6;1)*10+STXT(X1;8;1))*100)+STXT(X1;9;2)

9.6 Fonctions SQL

Ce chapitre présente de nombreux exemples de manipulation de dates, s'appuyant sur les fonctions standards de DB2 for i.

Récupération de la date système au format Date (2 syntaxes équivalentes) :

```
SELECT CURRENT DATE, CURDATE() FROM SYSIBM.SYSDUMMY1
```

Récupération de la date système au format YYYYMMDD :

```
SELECT
 YEAR(CURRENT DATE) * 10000 +
 MONTH(CURRENT DATE) * 100  +
 DAY(CURRENT DATE )
FROM SYSIBM.SYSDUMMY1 ;
```

Autre solution (plus élégante, et certainement plus performante) :

```
SELECT integer(replace(char(curdate(), ISO), '-', ''))
FROM SYSIBM.SYSDUMMY1 ;
```

Deux manières d'ajouter 1 semaine à la date système :

```
SELECT DATE(DAYS(CURRENT DATE)+7), CURRENT DATE + 7 DAYS
FROM SYSIBM.SYSDUMMY1 ;
```

Convertir une date SSAAMMJJ en JJMMSSAA en SQL :

```
Decimal(substring(digits(WDATE), 7, 2)
CONCAT substring(digits(WDATE), 5, 2 )
CONCAT substring(digits(WDATE), 1, 4 ),8 , 0)
```

Le numéro du jour dans le mois, pour une date donnée :

```
SELECT DAY(CURRENT DATE), DAY('2007-12-21')
FROM SYSIBM.SYSDUMMY1;
```

Le numéro du mois pour une date donnée :

```
SELECT MONTH(CURRENT DATE), MONTH('2007-12-21')
FROM SYSIBM.SYSDUMMY1 ;
```

Le numéro du jour de la semaine pour une date donnée. La fonction DAYOFWEEK considère que le dimanche est le 1er jour de la semaine, alors que la fonction DAYOFWEEK_ISO considère que c'est le lundi qui est le 1er jour de la semaine :

```
SELECT DAYOFWEEK(CURRENT DATE), DAYOFWEEK_ISO(CURRENT DATE)
FROM SYSIBM.SYSDUMMY1 ;
```

Le dernier jour du mois peut poser problème, selon la façon dont il est calculé :

```
SELECT
 date('2008-01-31') - 1 month as moins1,
 date('2008-01-31'),
 date('2008-01-31') + 1 month as plus1,
 date('2008-01-31') + 2 month as plus2
FROM SYSIBM.SYSDUMMY1 ;
```

MOINS1	DATE	PLUS1	PLUS2
31/12/07	31/01/08	29/02/08	31/03/08

La requête ci-dessus donne satisfaction car l'ajout ou la soustraction d'un mois ajuste bien chaque colonne calculée sur le dernier jour du mois considéré. Mais pour bien fonctionner, ce genre de calcul doit bien prendre comme référence un mois à 31 jours. Si on prenait pour référence le mois de février, ou un mois à 30 jours, le résultat serait tout autre. Exemple avec le

29/02/2008 :

```
SELECT
 date('2008-02-29') - 1 month as moins1,
 date('2008-02-29'),
 date('2008-02-29') + 1 month as plus1,
 date('2008-02-29') + 2 month as plus2
FROM SYSIBM.SYSDUMMY1 ;
```

```
MOINS1    DATE      PLUS1     PLUS2
29/01/08  29/02/08  29/03/08  29/04/08
```

ATTENTION : depuis la V5R3 et l'arrivée des fonctions DAYNAME et MONTHNAME, les requêtes ci-dessous (conservées pour l'exemple) peuvent être écrites plus simplement :

```
SELECT dayname(curdate()), monthname(curdate()) FROM qsqptabl
```

=> renvoie 'Lundi' et 'Janvier' pour le lundi 14 janvier 2008
=> attention : les valeurs retournées sont de type VARCHAR(100) avec un CCSID = 65535.

Avant la V5R3, pour récupérer le libellé du mois, on devait écrire :

```
SELECT CASE MONTH(CURRENT DATE)
WHEN  1 THEN 'Janvier'
WHEN  2 THEN 'Février'
WHEN  3 THEN 'Mars'
WHEN  4 THEN 'Avril'
WHEN  5 THEN 'Mai'
WHEN  6 THEN 'Juin'
WHEN  7 THEN 'Juillet'
WHEN  8 THEN 'Août'
WHEN  9 THEN 'Septembre'
WHEN 10 THEN 'Octobre'
WHEN 11 THEN 'Novembre'
ELSE         'Décembre'
END
FROM SYSIBM.SYSDUMMY1;
```

Le libellé du jour de la semaine pour une date donnée, en considérant le lundi comme le premier jour de la semaine :

```
SELECT CASE dayofweek_iso(current date)
WHEN 1 THEN 'Lundi'
WHEN 2 THEN 'Mardi'
WHEN 3 THEN 'Mercredi'
WHEN 4 THEN 'Jeudi'
WHEN 5 THEN 'Vendredi'
WHEN 6 THEN 'Samedi'
ELSE        'Dimanche'
```

```
END
FROM SYSIBM.sysdummy1;
```

Le libellé du jour de la semaine pour une date donnée, en considérant le dimanche comme le premier jour de la semaine :

```
SELECT CASE dayofweek(current date)
WHEN 1 THEN 'Dimanche'
WHEN 2 THEN 'Lundi'
WHEN 3 THEN 'Mardi'
WHEN 4 THEN 'Mercredi'
WHEN 5 THEN 'Jeudi'
WHEN 6 THEN 'Vendredi'
ELSE        'Samedi'
END
FROM SYSIBM.sysdummy1;
```

Obtenir le quantième pour une date donnée :

```
SELECT DAYOFYEAR(CURRENT DATE) FROM SYSIBM.SYSDUMMY1 ;
```

Ajouter à une date des jours, mois ou année pour en calculer une autre :

Par exemple, quelle est la date du jour + 2 ans + 3 mois + 5 jours :

```
SELECT CURRENT DATE + 2 YEARS + 3 MONTHS + 5 DAYS
FROM SYSIBM.SYSDUMMY1 ;
```

Nombre de jours depuis le 1er janvier de l'an 1 :

Par exemple la requête suivante donne 1 :

```
SELECT DAYS('0001-01-01') FROM SYSIBM.SYSDUMMY1;
```

Par exemple la requête suivante donne 732302 :

```
SELECT DAYS('2005-12-22') FROM SYSIBM.SYSDUMMY1;
```

Calculer le nombre de jours calendaires entre deux dates :

Par exemple la requête suivante donne 45 jours :

```
SELECT DAYS('2005-12-22') - DAYS('2005-11-07')
FROM SYSIBM.SYSDUMMY1 ;
```

Calculer le numéro de la semaine pour une date donnée

Par exemple la requête suivante donne la semaine 50 :

```
SELECT DAYOFYEAR('2005-12-22') / 7
FROM SYSIBM.SYSDUMMY1 ;
```

Utiliser les fonctions dates sur une date stockée au format numérique 8.0, en passant la date au format 8.0 en caractère avec séparateur.

Par exemple la requête suivante extrait le numéro de semaine pour chaque date 'DATE1' :

```
SELECT DAYOFYEAR(
 SUBSTRING(CHAR(DATE1), 1, 4) CONCAT '-' CONCAT
 SUBSTRING(CHAR(DATE1), 5, 2) CONCAT '-' CONCAT
```

```
 SUBSTRING(CHAR(DATE1), 7, 2)
 ) / 7
 FROM TABLE1 ;
```

Calculer le numéro de trimestre d'une date donnée :
QUARTER, convertit une date, un timestamp en une valeur entière représentant le trimestre de l'année concernée par la date :
1 pour 1er trimestre
2 pour 2nd trimestre
3 pour 3ème trimestre
4 pour 4ème trimestre

```
SELECT QUARTER('2007-04-11-11.59.20.918664') FROM SYSIBM.SYSDUMMY1 ;
```
-> Donne comme résultat : 2

```
SELECT QUARTER(DATE('2007-04-11-11.59.20.918664') + 4 MONTHS)
 FROM SYSIBM.SYSDUMMY1 ;
```
-> Donne comme résultat : 3

```
SELECT quarter(CURRENT DATE) FROM SYSIBM.SYSDUMMY1;
```
-> Donne comme résultat : 1 pour une date au 19/01/2007

Conversion du format « time » en format alphanumérique ou numérique

- solution 1 :

```
SELECT curtime(), replace(char(curtime()), ':', '')
FROM SYSIBM.SYSDUMMY1;

CURTIME ( )  REPLACE
 11:51:13    115113
```

- solution 2 :

```
SELECT curtime(), integer(replace(char(curtime()), ':', ''))
FROM SYSIBM.SYSDUMMY1;

CURTIME ( )         INTEGER
 11:51:38           115.138
```

- solution 3 :

```
SELECT curtime(),
  substr(char(curtime()), 1, 2) concat
  substr(char(curtime()), 4, 2) concat
  substr(char(curtime()), 7, 2)
FROM SYSIBM.SYSDUMMY1;
```

```
CURTIME ( )  Expression alphanum
 11:53:54           115354
```

9.7 Considérations de performances

Il est important de noter qu'aucun index ne sera utilisé pour une colonne si cette dernière fait partie d'une expression arithmétique.
Par exemple, le prédicat dans l'expression ci-dessous n'est pas indexable :

```
SELECT * FROM FICHIER a
WHERE a.MADATE - 10 DAYS = :VAR_HOTE ;
```

On peut contourner le problème en disposant différemment les termes de l'expression arithmétique, de la façon suivante :

```
SELECT * FROM FICHIER a
WHERE a.MADATE = :VAR_HOTE + 10 DAYS ;
```

Les 2 approches sont strictement équivalentes d'un point de vue logique, mais la seconde peut permettre d'obtenir de bien meilleures performances dans de nombreux cas, car elle permet à l'optimiseur SQL de s'appuyer sur un index exploitant la colonne MADATE (de type DATE), si elle existe.

10. SQL dans les programmes RPG

10.1 inclusion de code en RPG et Adelia

L'inclusion de code SQL dans un programme RPG se fait de la façon suivante :

En RPG Fixe :

```
C/EXEC SQL
C+ update matable set ...
C/END-EXEC
```

En RPG Free :

```
  EXEC SQL
    update matable set ... ;
```

En Adelia :

```
DEBUT_SQL
   + update matable set ...
FIN_SQL
```

Un source RPG contenant des requêtes SQL devra forcément être de type SQLRPGLE pour pouvoir être compilé.
En Adelia, c'est le générateur RPG qui détermine implicitement le type de source à générer lors de la compilation en fonction de son contenu (le développeur n'a pas à s'en préoccuper).

10.2 Directives d'exécution

Il est possible de personnaliser certaines directives d'exécution du code SQL par l'intermédiaire d'une ligne de code SQL telle que celle-ci-dessous :

```
DEBUT_SQL
  + SET OPTION DATFMT = *ISO , TIMFMT = *ISO , CLOSQLCSR = *ENDMOD
FIN_SQL
```

On pourrait par exemple spécifier le niveau d'isolement souhaité en modifiant l'option associée au paramètre COMMIT. On pourrait écrire cela de la façon suivante :

```
IF <condition> ;
  EXEC SQL
    SET OPTION COMMIT = *NONE ;
ELSE
  EXEC SQL
    SET OPTION COMMIT = *CHG ;
END IF ;
```

Les différents paramètres modifiables par l'intermédiaire de SET OPTION sont les suivants :

ALWBLK	DECRESULT	RDBCNNMTH
ALWCPYDTA	DFTRDBCOL	SQLCA
CLOSQLCSR	DLYPRP	SQLCURRULE
CNULRQD	DYNDFTCOL	SQLMODE
COMMIT	DYNUSRPRF	SQLPATH
COMPILEOPT	EVENTF	SRTSEQ
DATFMT	LANGID	TGTRLS
DATSEP	NAMING	TIMFMT
DBGVIEW	OPTLOB	TIMSEP
DECMPT	OUTPUT	USRPRF

On retrouve la plupart des paramètres du SET OPTION sur la commande RUNSQLSTM, il est donc possible de se reporter à la documentation de cette commande pour obtenir des précisions sur le fonctionnement des différents paramètres et sur leurs valeurs autorisées respectives.

Présentation rapide de quelques paramètres parmi les plus utilisés (avec la liste des valeurs autorisées pour chacun d'entre eux) :

CLOSQLCSR : *ENDACTGRP *ENDMOD	**COMMIT :** *CHG *UR *CS *ALL *RS *NONE *NC *RR	**DATFMT :** *JOB *USA *ISO *EUR *JIS *MDY *DMY *YMD *JUL
DATSEP : *JOB / . , - *BLANK	**DECMPT :** *JOB *SYSVAL *PERIOD *COMMA	**TIMFMT :** *HMS *USA *ISO *EUR *JIS
TIMSEP : *JOB : . , *BLANK		

10.3 Gestion des erreurs - Les bases

La data-structure SQLCA est incluse automatiquement, par le précompilateur SQL, dans les caractéristiques de définition du RPG généré (cartes D). Elle permet de récupérer des informations relatives à l'exécution d'une requête SQL. Il est par exemple possible de récupérer le nombre d'enregistrements mis à jour (SQLER3) ou de détecter la fin d'un fichier (SQLCODE = 100).

Du fait de l'inclusion automatique par le précompilateur, il est inutile de coder l'ordre d'inclusion de SQLCA dans le programme Adelia (idem en RPG). A titre purement informatif, la déclaration manuelle de SQLCA s'écrirait de la façon suivante :

Déclaration explicite (superflue) :

En Adelia :	En RPG :

`DEBUT_SQL`
`  + INCLUDE SQLCA`
`FIN_SQL`
	`C/EXEC SQL`
	`C+ INCLUDE SQLCA`
	`C/END-EXEC`

Définition de la DS SQLCA en RPG ILE :

```
D*      SQL Communications area
D SQLCA           DS
D  SQLCAID                       8A   INZ(X'0000000000000000')
D  SQLAID                        8A   OVERLAY(SQLCAID)
D  SQLCABC                      10I 0
D  SQLABC                        9B 0 OVERLAY(SQLCABC)
D  SQLCODE                      10I 0
D  SQLCOD                        9B 0 OVERLAY(SQLCODE)
D  SQLERRML                      5I 0
D  SQLERL                        4B 0 OVERLAY(SQLERRML)
D  SQLERRMC                     70A
D  SQLERM                       70A   OVERLAY(SQLERRMC)
D  SQLERRP                       8A
D  SQLERP                        8A   OVERLAY(SQLERRP)
D  SQLERR                       24A
D   SQLER1                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER2                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER3                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER4                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER5                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLER6                       9B 0 OVERLAY(SQLERR:*NEXT)
D   SQLERRD                     10I 0 DIM(6)  OVERLAY(SQLERR)
D  SQLWRN                       11A
D   SQLWN0                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN1                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN2                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN3                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN4                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN5                       1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN6                       1A   OVERLAY(SQLWRN:*NEXT)
```

```
D   SQLWN7                        1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN8                        1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWN9                        1A   OVERLAY(SQLWRN:*NEXT)
D   SQLWNA                        1A   OVERLAY(SQLWRN:*NEXT)
D  SQLWARN                        1A   DIM(11) OVERLAY(SQLWRN)
D  SQLSTATE                       5A
D  SQLSTT                         5A   OVERLAY(SQLSTATE)
D*  End of SQLCA
```

Les variables de SQLCA les plus utiles sont indiquées ci-dessous. Elles doivent être utilisées juste après l'exécution de l'ordre SQL concerné.

SQLCODE ou SQLCOD : Permet de détecter les erreurs (en Adelia, SQLCODE est stocké dans le mot réservé *SQLCODE)
- Si SQLCOD = 0 alors pas d'erreur (ni d'avertissement), la requête SQL a été exécutée avec succès
- Si SQLCOD = +100 alors Fin de fichier détectée sur la dernière requête SQL traitée
- Si SQLCOD > 0 et <> de +100 alors Avertissement, néanmoins la requête SQL a bien été exécutée
- Si SQLCOD < 0 Erreur détectée, l'ordre SQL n'a pas été exécuté.

SQLERRMC : Texte du message SQL d'erreur si SQLCODE < 0.

SQLERRML : longueur significative de SQLERRMC

SQLERRD ou SQLERR : DS de traitements des erreurs et avertissements, composée de 6 zones (6 fois 4 octets binaires) de SQLER1 à SQLER6
- SQLER1 contient le n° de message CPFxxxx (si SQLCODE < à 0)
- SQLER2 contient le N° message CPDxxxx (si SQLCODE < à 0)
- SQLER3 contient le nombre d'enregistrements

SQLWRN ou SQLWARN : zone de 11 caractères
- SQLWRN0 contient 'W' dans le cas où SQLCODE > 0 et <> + 100

SQLSTT ou SQLSTATE : zone de 5 caractères :
- les 2 premiers caractères contiennent un code « classe » (décrit plus loin dans ce chapitre)
- les 3 caractères suivants (positions 3 à 5) précisent le type d'anomalie rencontré

Le nombre d'erreurs et d'avertissements renvoyés par SQLCODE est trop important pour être indiqué ici (on pourra se référer en cas de besoin à la documentation IBM). La liste complète des codes classes définissant le SQLSTATE étant plus réduire, elle est indiquée ci-dessous à titre d'information.

Code classe	Type d'erreur
00	Instruction exécutée avec succès
01	Avertissement (instruction exécutée malgré tout)
02	Pas de donnée (enregistrement non trouvé, ou fin de fichier)

07	Erreur sur SQL dynamique
08	Erreur (exception) sur Connection
16	Dispositif non supporté
21	Violation de cardinalité
22	Erreur (exception) sur donnée
23	Violation de contrainte
24	Etat de curseur invalide
26	Syntaxe SQL invalide
2D	Arrêt de transaction invalide
34	Nom de curseur invalide
37	Erreur de syntaxe
39	Appel de fonction externe en erreur
40	Echec d'exécution séquentielle
42	Violation d'accès
44	Violation de clause WITH CHECK OPTION
51	Etat d'application invalide
52	Nom dupliqué ou indéfini
53	Opérandes invalids ou spécifications contradictoires
54	Dépassement de capacité de SQL
55	Objet dans un état non acceptable
56	Restrictions diverses de SQL
57	Resource indisponible ou intervention de l'opérateur requise
58	Système en erreur

Si on compare le contenu de SQLSTATE avec celui de SQLCODE, on note que le code classe de SQLSTATE :
- contient '00' si SQLCODE = 0
- contient '01' si SQLCODE > 0 et <> de 100
- contient '02' si SQLCODE = +100 (dans ce cas précis SQLSTATE contient '02000').
- contient une valeur différente de '00', '01' et '02' si SQLCODE < 0

Il est également intéressant de noter qu'il n'y a pas forcément un SQLSTATE pour un SQLCODE. Certains SQLCODE peuvent pointer vers le même SQLSTATE. L'échantillon de tableau ci-dessous, extrait d'un ouvrage en langue anglaise, et trié par SQLSTATE, donne un aperçu de différents cas que l'on peut rencontrer :

SQLSTATE	SQLCODE	Description
00000	000	The SQL statement finished successfully.
01004	+445	Value has been truncated by a CAST function.
01005	+236	The value of SQLN in the SQLDA should be at least as large as the number of columns that are being described.
01005	+238	At least one of the columns being described is a LOB, so additional space is required for extended SQLVAR entries.
01005	+239	At least one of the columns being described is a distinct type, so additional space is required for extended SQLVAR entries.
01514	+162	Named tablespace placed in check pending status.
01515	+304	Value cannot be assigned to host variable because it is out of range for the data type.
01516	+558	Already granted to PUBLIC so WITH GRANT OPTION not applicable.

01518	+625	Table definition marked incomplete because primary key index was dropped.
01519	+802	Data exception error caused by data overflow or divide exception.
01520	+331	String cannot be translated so it has been assigned to NULL.

SQLCODE ou SQLSTATE, lequel faut-il utiliser ?
Entre SQLCODE, SQLWARN et SQLSTATE, il est vrai qu'il y a de quoi se perdre. Si l'on exclut SQLWARN du périmètre, vaut-il mieux « checker » SQLCODE ou SQLSTATE après l'exécution d'une requête ?

En ce qui concerne SQLCODE, en dehors des valeurs 0 et 100 qui sont communes à l'ensemble des produits de la famille DB2, les autres valeurs définissant les avertissements et les erreurs ne sont pas homogènes.

En revanche, SQLSTATE est conforme à la norme ISO/ANSI SQL92. Il en résulte que le tableau des différentes valeurs possibles pour SQLSTATE est strictement identique sur les différentes versions de DB2. SQLSTATE offre donc une excellente portabilité du code SQL, sur les différentes plateformes DB2.

L'utilisation de SQLCODE pose un problème. Dans la documentation d'Adelia, HARDIS préconise de tester la bonne exécution d'une requête par l'un des tests suivants :
- 1ère solution :
```
SI *SQLCODE = *NORMAL
```
- 2ème solution (strictement équivalente à la première) :
```
SI *SQLCODE = 0
```
- 3ème solution (préconisée par Hardis dans les boucles de lecture) :
```
TANT_QUE *SQLCODE <> 100
```

Si l'on considère que toutes les valeurs de SQLCODE supérieures à zéro et différentes de 100 correspondent à des avertissements non bloquants pour lesquels la requête s'est malgré tout exécutée, il est préférable de tester la bonne exécution de la requête au moyen d'un test plus complet, comme dans l'exemple ci-dessous :
```
FETCH …
TANT_QUE *SQLCODE >= 0 et *SQLCODE <> 100
  …
  FETCH …
REFAIRE
```
On recommandera d'utiliser le principe ci-dessus aussi bien sur les SI que sur les TANT_QUE.

Conclusion concernant l'utilisation de SQLCODE et SQLSTATE :

- Approche privilégiant SQLCODE :

Si la portabilité du code SQL n'est pas la préoccupation majeure des équipes de développement, elles pourront privilégier l'indicateur SQLCODE en se basant sur les exemples ci-dessous :

```
En Adelia :
```

```
SI *SQLCODE >= 0 ET *SQLCODE <> 100
   *-- OK (requête exécutée)
SINON_SI *SQLCODE = 100
   *-- EOF (fin de fichier)
SINON
   *-- KO (erreur d'exécution)
FIN
```

En RPG :

```
C                   IF        SQLCOD >= 0 AND SQLCOD <> 100
C* OK (requête exécutée)
C                   ELSEIF    SQLCOD = 100
C* EOF (fin de fichier)
C                   ELSE
C* KO (erreur d'exécution)
C                   END
```

Remarques :

- Le test qui est inclus dans le SINON_SI (ELSIF) sur SQLCODE = 100 est bien évidemment facultatif. Si aucune action particulière ne doit être effectuée sur une « fin de fichier », alors ce test peut être omis.
- Le dernier SINON (ELSE) correspond à un SQLCODE négatif, il devrait donc entraîner le débranchement vers une procédure de traitement d'erreur, visant au minimum à consigner dans un fichier de log l'anomalie rencontrée. Ce point devra faire l'objet d'une attention particulière lors de la définition de normes et standards de développement.

- Approche privilégiant SQLSTATE :

Si les équipes de développement souhaitent privilégier la norme ISO/ANSI SQL92, elles pourront utiliser l'approche suivante :

En Adelia :

```
*-- SQLSTATE et WSQLCLASS devront être déclarées dans l'environnement
*    de données du programme Adelia, ou à l'intérieur du code Adelia
*    au moyen des instructions suivantes :
DECLARER SQLSTATE;SQLSTATE 5 *NODEF
DECLARER WSQLCLAS;WSQLCLAS 2

… requête SQL …

WSQLCLAS = &EXTRACTION(SQLSTATE ; 1 ; 2)
SI WSQLCLAS = '00';'01'
   * OK (requête exécutée)
SINON_SI WSQLCLAS = '02'
   * EOF (fin de fichier)
SINON
   * KO (erreur d'exécution)
FIN
```

En RPG :

```
 * CARTES D (seule WSQLCLAS doit être declarée, SQLSTATE étant implicite)
DWSQLCLAS         S              2

… requête SQL …

C                   EVAL      WSQLCLAS = %SUBST(SQLSTATE:1:2)
C                   IF        WSQLCLAS = '00' OR WSQLCLAS = '01'
C* OK (requête exécutée)
C                   ELSEIF    WSQLCLAS = '02'
C* EOF (fin de fichier)
C                   ELSE
C* KO (erreur d'exécution)
C                   END
```

Remarques :

- Les remarques de la page précédente sont également valables pour les exemples présentés ci-dessus.
- Adelia offre un mot réservé *SQLSTATE, mais ce mot réservé n'est pas disponible pour les développements 5250 (il est en revanche disponible pour les développements de type Visual et Web Adelia). L'absence de ce mot réservé pour le développement 5250 n'est toutefois pas un réel problème, puisque l'on peut contourner le problème en déclarant une variable SQLSTATE de 5 caractères en *NODEF comme dans l'exemple ci-dessus.
- Il est préférable de préfixer la variable SQLCLASS avec un caractère (« W » dans l'exemple ci-dessus) pour éviter tout risque d'incompatibilité avec une évolution toujours possible du précompilateur SQL, qui pourrait inclure une définition explicite du code classe.

10.4 Gestion des erreurs - Get Diagnostics

La fonction GET DIAGNOSTICS, apparue en V5R3, fonctionne aussi bien en RPG qu'en ADELIA (ainsi que dans les procédures stockées), et permet donc d'envisager une solution homogène pour la gestion des erreurs pour la plupart des langages embarquant du code SQL.

Exemple ci-dessous avec un petit programme SQLRPGLE :

```
H DECEDIT('0,') DATEDIT(*YMD)  DEBUG
 *
DWVAR             S             10  0
DWVAR2            S             10  0
Dnberror          S              7  0
Dwi               S              7  0
Dcstname          S             80
Dsqlmsg           S             80
Dsqlsttsav        S              4
 *
C/Exec SQL
C+ VALUES (SELECT MIN(NUMCLI), MAX(NUMCLI) FROM CLIENT) INTO :WVAR , :WVAR2
C/End-Exec
 /free
      if %subst(sqlState:1:2) <> '00' ;
          Exsr SQLError ;
      endif ;
 /end-free
C                   SETON                                        LR
 ******************************************
C     SQLERROR      BEGSR
 *
 *  Stockage du code statut de l'erreur
 /free
       sqlsttsav = %subst(sqlState:1:4) ;
 /end-free
 *  Récupération du nombre d'erreurs SQL
C/EXEC SQL
C+   GET DIAGNOSTICS :nberror = NUMBER
C/END-EXEC
*  Boucle de traitement des différentes erreurs SQL
 /free
       for wi = 1 to nberror ;
 /end-free
 *  Récupération des caractéristiques de chaque erreur
C/EXEC SQL
C+   GET DIAGNOSTICS CONDITION :wi
C+     :cstname = CONSTRAINT_NAME,
C+     :sqlMsg  = MESSAGE_TEXT
C/END-EXEC
 *  Dump pour afficher le détail de l'erreur
C                   DUMP
 /free
       endfor ;
```

```
 /end-free
C                   ENDSR
```

Dans l'exemple ci-dessus, on appelle la sous-routine SQLERROR si le SQLSTATE de la dernière requête SQL réalisée est différent de « 00 ». Cette sous-routine effectue plusieurs actions :

- sauvegarde dans une variable SQLSTTSAV du code statut de l'erreur (car il sera écrasé par les requêtes SQL suivantes),
- récupération par GET DIAGNOSTICS du nombre d'erreurs SQL dans la variable NBERROR (selon le type de requête en erreur, il sera de valeur 1 ou supérieure),
- traitement d'une boucle de récupération de chaque erreur dans des variables de travail (CSTNAME, SQLMSG) puis DUMP pour générer un spoule pour chaque erreur.

A noter que l'on aurait pu stocker les erreurs dans un fichier, ou envoyer un message dans une log, etc…

Evolutions GET DIAGNOSTIC en V7 (SF99601 level 21, SF99701 level 11) :

- GET DIAGNOSTICS ma_variable = ROW_COUNT
  retourne le nombre de lignes insérées suite à CREATE TABLE ou DECLARE GLOBAL TEMPORARY TABLE, **WITH DATA**
- GET DIAGNOSTICS CONDITION 1 ma_variable = MESSAGE_TEXT
  retourne le texte du dernier message d'erreur **renvoyé par une fonction (UDF) ou une fonction TABLE (UDTF) avec** *parameter style SQL.*

A noter :

GET DIAGNOSTICS fonctionne très bien dans les procédures stockées DB2. L'instruction doit être placée immédiatement derrière l'instruction SQL à tracer, quel que soit son type.

Exemple :

```
delete from TAB_CLIENT ;
GET DIAGNOSTICS V_NBR_ENR = ROW_COUNT;
```

La variable V_NBR_ENR doit avoir été déclarée au préalable, via la syntaxe suivante :

```
DECLARE V_NBR_ENR INTEGER DEFAULT 0;
```

10.5 SELECT … INTO …

Les exemples ci-dessous ont pour particularité de combiner l'utilisation des sous-requêtes scalaires de type « full select » avec les syntaxes suivantes :

- SELECT … INTO *:variables_hôte(s)* FROM ….
- SET *:variable_hôte* = (*sous requête*)
- VALUES (sous-requête) INTO *:variables_hôte(s)*

Pour de plus amples précisions sur les requêtes de type « full select », voir aussi le chapitre dédié à ce sujet dans le document « Doc_DB2SQL_Bases ».

La requête ci-dessous consiste à lire une ligne unique de la table T1, et à renvoyer le contenu d'une colonne C2 dans une variable hôte V2 :

```
V2 = *BLANK     * ou V2 = 0 si numérique
DEBUT_SQL
 + SELECT C2
 +   INTO :V2
 + FROM T1
 + WHERE C1 = :V1
FIN_SQL
```

En fait la forme d'écriture ci-dessus, basée sur l'association des prédicats SELECT et INTO, peut s'écrire sous 2 autres formes :

- 1ère forme : avec la clause SET

```
V2 = *BLANK     * ou V2 = 0 si numérique
DEBUT_SQL
 + SET :V2 = (SELECT C2 FROM T1 WHERE C1 = :V1)
FIN_SQL
```

- 2ème forme : avec la clause VALUES … INTO

```
V2 = *BLANK     * ou V2 = 0 si numérique
DEBUT_SQL
 + VALUES (SELECT C2 FROM T1 WHERE C1 = :V1) INTO :V2
FIN_SQL
```

Le développeur a toute liberté dans le choix de l'écriture. S'il a besoin de récupérer plusieurs valeurs en sortie d'une requête, il pourra utiliser l'une des 2 écritures suivantes :

- première forme « traditionnelle » :

```
V4 = *BLANK     * ou V4 = 0 si numérique
V5 = *BLANK     * ou V5 = 0 si numérique
DEBUT_SQL
 + SELECT C4, C5
 +   INTO :V4, :V5
 + FROM T1
 + WHERE C1 = :V1 AND C2 = :V2
FIN_SQL
```

- seconde forme utilisant la clause VALUES … INTO :

```
   V4 = *BLANK    * ou V4 = 0 si numérique
   V5 = *BLANK    * ou V5 = 0 si numérique
   DEBUT_SQL
    + VALUES (SELECT C4, C5 FROM T1 WHERE C1 = :V1 AND C2 = :V2)
    +   INTO :V4 , :V5
   FIN_SQL
```

Autre exemple démontrant la facilité de manipulation des dates, cette fois-ci dans la syntaxe SQL RPG :

```
H DECEDIT('0,') DATEDIT(*YMD)  DEBUG
 *
DWVARO1           S               d
DWVARO2           S               d
DWVARI1           S              2s 0 inz(1)
DWVARI2           S              2s 0 inz(3)
 *
C/Exec SQL
C+ VALUES (SELECT CURRENT DATE,
C+  CURRENT DATE + :WVARI1 MONTHS - :WVARI2 DAYS
C+  FROM QSQPTABL)
C+ INTO :WVARO1 , :WVARO2
C/End-Exec
C                   DUMP
C                   SETON                                        LR
```

Si l'on exécute ce code le 22/06/2009, le Dump produit le résultat suivant :

```
 WVARI1               ZONED(2,0)          01.
 WVARI2               ZONED(2,0)          03.
 WVARO1               DATE(10)            '2009-06-22'
 WVARO2               DATE(10)            '2009-07-19'
```

ATTENTION : On rappelle que ces différentes techniques ne fonctionnent qu'à la seule condition que les sous-requêtes considérées ne renvoient qu'une seule ligne (principe du « full select »). Si la sélection définie dans la clause WHERE ne permet pas de récupérer à coup sûr une seule ligne, il est possible dans certains cas de contourner le problème en ajoutant à la requête la clause « FETCH FIRST 1 ROW ONLY » comme dans l'exemple ci-dessous :

```
   V2 = *BLANK    * ou V2 = 0 si numérique
   DEBUT_SQL
    + SELECT C2
    +   INTO :V2
    + FROM T1
    + WHERE C1 = :V1
    + FETCH FIRST 1 ROW ONLY
   FIN_SQL
```

La technique présentée ci-dessus peut être utilisée également pour réaliser des comptages (ou encore des sommes ou des moyennes), à la condition de ne renvoyer, là-encore, qu'un seul enregistrement en sortie (l'utilisation de la clause GROUP BY est à proscrire dans ce cas).

Exemple ci-dessous avec la fonction de comptage COUNT :

```
*-- Initialisation de la variable de comptage
VN_NBR = 0
*-- Requête de comptage
DEBUT_SQL
 + SELECT COUNT(*)
 +   INTO :VN_NBR
 + FROM T1
 + WHERE C1 = :V1
FIN_SQL
*
SI *SQLCODE >= 0 ET *SQLCODE <> 100
   SI VN_NBR > 0
      … traitement fonctionnel …
   FIN
FIN
```

IMPORTANT : les variables hôtes adressées par la clause INTO peuvent être regroupées dans une DS (appelée « structure hôte » en jargon SQL). La clause INTO peut donc adresser une structure hôte ce qui offre 2 avantages :

- une simplification dans l'écriture de la requête SQL
- de meilleures performances d'exécution.

Une autre application pratique des requêtes de type « full select » consiste à les utiliser dans les algorithmes de contrôle de validité de zone, et de récupération de libellés. Par exemple, dans le pavé VERIFICATION d'un programme Adelia, au lieu d'écrire ceci :

```
SI ZZZ_NUM_CLI = *BLANK
   * Contrôle de type « Zone obligatoire »
   ZZZ_LIB = *BLANK
   PREPARER_MSG 0001 ZZZ_NUM_CLI
   ANOMALIE
SINON
   * Lecture du fichier TABCLI pour récupération du libellé client
   LIRE TABCLI
   SI TABCLI EXISTE
      ZZZ_LIB = LIBCLI
   SINON
      * Code erroné, non trouvé dans la table TABCLI
      ZZZ_LIB = *BLANK
      PREPARER_MSG 0002 ZZZ_NUM_CLI
      ANOMALIE
   FIN
FIN
```

On peut écrire ceci :

```
SI ZZZ_NUM_CLI = *BLANK
   * Contrôle de type « Zone obligatoire »
   ZZZ_LIB = *BLANK
   PREPARER_MSG 0001 ZZZ_NUM_CLI
   ANOMALIE
SINON
   * Initialisation de l'indicateur de recherche SQL
   W_SQL_TROUVE = *BLANK
   * Initialisation du libellé affiché à l'écran
   ZZZ_LIB = *BLANK
   DEBUT_SQL
      + VALUES (SELECT '1', LIBCLI FROM TABCLI WHERE
      +   CODSOC = :ZZZ_COD_SOC AND CODCLI = :ZZZ_NUM_CLI)
      +   INTO :W_SQL_TROUVE , :ZZZ_LIB
   FIN_SQL
   SI W_SQL_TROUVE <> '1' OU SQLCODE <> 0
      * Code erroné, non trouvé dans la table TABCLI
      PREPARER_MSG 0002 ZZZ_NUM_CLI
      ANOMALIE
   FIN
FIN
```

CONCLUSION : les techniques présentées ici offrent à mon avis deux avantages majeurs :

- elles permettent de s'affranchir des problèmes de niveau de version, inhérents à l'utilisation d'ordres de lecture natifs. Ainsi, si la structure de la table TABCLI est modifiée, alors les programmes qui accèdent à cette table via SQL - simplement pour contrôler un identifiant et/ou récupérer un libellé - n'ont plus besoin d'être recompilés. On réduit ainsi le nombre d' « adhérences » entre les programmes et les fichiers DB2, et on gagne en réactivité au niveau du processus de développement. Bien évidemment, ce que je viens d'écrire n'est valable que dans les cas où la modification de la table TABCLI ne concerne pas les colonnes utilisées dans la sous-requête scalaire.
- même si la table SQL utilisée dans la sous-requête contient beaucoup de colonnes et possède un gros buffer, le programme qui utilise cette sous-requête ne « voit » que les colonnes dont il a réellement besoin. Cela permet de garantir d'excellentes performances (à condition bien évidemment, que la table utilisée dans la sous-requête possède un index adapté à la sélection demandée).

10.6 Curseur SQL statique

Exemple de curseur SQL incrémentiel défini avec une requête statique :

```
DS DSEXPD W_EXIDEN1 W_EXIDEN2 W_EXINENL W_EXIMONL W_EXDCXDX W_EXDCXGE
*
*-- DECLARATION DU CURSEUR
DEBUT_SQL
   + DECLARE C_LIGNE CURSOR FOR
   + SELECT EXIDEN1, EXIDEN2, EXINENL,
   +        EXIMONL, EXDCXDX, EXDCXGE
   + FROM FICTST
   + ORDER BY EXIDEN1
   + FOR FETCH ONLY    -- cette clause est documentée dans la suite de cette doc
FIN_SQL
*
*-- OUVERTURE DU CURSEUR
DEBUT_SQL
   + OPEN C_LIGNE
FIN_SQL
*
*-- PREMIERE LECTURE DU CURSEUR
TRAITER_PROC LECCUR
*
*-- BOUCLE DE TRAITEMENT DU CURSEUR
TANT_QUE *SQLCODE >= 0 ET *SQLCODE <> 100
   TRAITER_PROC TRTENR  * procédure non définie à adapter selon le traitement
   TRAITER_PROC LECCUR
REFAIRE
*
*-- FERMETURE DU CURSEUR
DEBUT_SQL
   + CLOSE C_LIGNE
FIN_SQL
TERMINER
************************
DEBUT_PROCEDURE LECCUR
   *-- PROCEDURE DE LECTURE DU CURSEUR
   DEBUT_SQL
      + FETCH C_LIGNE INTO :DSEXPD
   FIN_SQL
FIN_PROCEDURE
```

Quelques remarques :

- Avec un curseur incrémentiel, la lecture se fait du début à la fin de la séquence d'enregistrements sélectionné par la requête SQL. L'ordre FETCH suffit pour passer d'une ligne à l'autre, on pourrait éventuellement utiliser l'ordre FETCH suivi de l'option NEXT, mais ce n'est pas indispensable, et même dangereux car dans le cas du premier FETCH, l'option NEXT aurait pour effet de « sauter » la première ligne et de se positionner directement sur la seconde. Il est donc recommandé de n'utiliser les options du FETCH que sur les curseurs non incrémentiels (cf. exemple de la page suivante).

- de manière à éviter toute redondance de code, on recommandera de placer le FETCH dans une

procédure (appelée LECCUR dans l'exemple ci-dessus), cette dernière étant appelée 2 fois (une fois avant le TANT_QUE, une fois à l'intérieur du TANT_QUE).

- le second exemple de programme SQLRPGLE fourni en annexe effectue un traitement strictement identique à l'exemple Adelia présenté dans ce chapitre.

- dans l'exemple présenté ci-dessus, on a utilisé une technique d'optimisation consistant à regrouper les colonnes destinataires du FETCH dans une DS. Cela permet de simplifier l'écriture du FETCH (qui n'adresse qu'une variable regroupant les différentes colonnes transmises par SQL).

10.7 Curseur SQL dynamique

Le développeur peut rencontrer des cas où la technique du curseur statique se révèle trop contraignante. C'est notamment le cas si les conditions du SELECT sont très fluctuantes (par exemple en fonction de critères saisis par l'utilisateur), ou s'il est nécessaire de paramétrer le tri (par ORDER_BY) selon différents critères (par exemple : paramétrage filiale, ou saisie d'utilisateur, etc…). La technique du SQL dynamique se révèle alors beaucoup plus pratique, et elle doit être privilégiée, même si elle fait perdre quelques uns des avantages du SQL statique (cf. chapitre précédent).

Dans l'exemple ci-dessous, on a repris la requête SQL du chapitre précédent, dont on a redéfini la déclaration de manière dynamique :

```
DS DSEXPD W_EXIDEN1 W_EXIDEN2 W_EXINENL W_EXIMONL W_EXDCXDX W_EXDCXGE
*
*-- DEFINITION DE LA REQUETE DANS UNE VARIABLE
W_REQUETE = 'Select EXIDEN1, EXIDEN2, EXINENL, '
W_REQUETE = W_REQUETE /// ' EXIMONL, EXDCXDX, EXDCXGE  '
W_REQUETE = W_REQUETE /// ' From FICTST Order By EXIDEN1 '
W_REQUETE = W_REQUETE /// ' For FETCH ONLY '
*
*-- PREPARATION DE LA REQUETE
DEBUT_SQL
   + PREPARE REQ1 FROM :W_REQUETE
FIN_SQL
*
*-- DECLARATION DU CURSEUR (incrémentiel)
DEBUT_SQL
   + DECLARE C_LIGNE CURSOR FOR REQ1
FIN_SQL
*-- OUVERTURE DU CURSEUR
DEBUT_SQL
   + OPEN C_LIGNE
FIN_SQL
*-- PREMIERE LECTURE DU CURSEUR
TRAITER_PROC LECCUR
*-- BOUCLE DE TRAITEMENT DU CURSEUR
TANT_QUE *SQLCODE >= 0 ET *SQLCODE <> 100
   TRAITER_PROC LECCUR
REFAIRE
*-- FERMETURE DU CURSEUR
DEBUT_SQL
   + CLOSE C_LIGNE
FIN_SQL
*
TERMINER
************************
DEBUT_PROCEDURE LECCUR
   *
   *-- PROCEDURE DE LECTURE DU CURSEUR
   DEBUT_SQL
      + FETCH C_LIGNE INTO :DSEXPD
   FIN_SQL
   *
FIN_PROCEDURE
```

L'intérêt principal de la technique présentée ci-dessus réside dans la souplesse qu'apporte la définition de la variable W_Requete. Les clauses WHERE et ORDER_BY peuvent être définies par concaténation de portions de chaînes, en fonction de paramétrages et/ou de saisies d'utilisateurs.

Points importants :
- les remarques formulées pour les curseurs SQL statiques s'appliquent également aux curseurs SQL dynamiques. On rappellera également que la technique du curseur non incrémentiel (SCROLL CURSOR) fonctionne également sur les curseurs SQL dynamiques et qu'il est vivement recommandé d'ouvrir le curseur en lecture seule (FOR FETCH ONLY) si aucune mise à jour n'est requise en cours de traitement.

- il est possible d'utiliser la technique des requêtes paramétrées (avec ?) à l'intérieur de curseurs SQL dynamiques. Si la requête de la page précédente avait contenu une sélection à l'intérieur de sa clause WHERE, on aurait pu écrire ceci :

```
*-- DEFINITION DE LA REQUETE DANS UNE VARIABLE
W_REQUETE = 'Select EXIDEN1, EXIDEN2, EXINENL, '
W_REQUETE = W_REQUETE /// ' EXIMONL, EXDCXDX, EXDCXGE  '
W_REQUETE = W_REQUETE /// ' From FICTST Order By EXIDEN1 '
W_REQUETE = W_REQUETE /// ' Where EXIDEN1 = ? and EXIDEN2 = ? '
W_REQUETE = W_REQUETE /// ' For FETCH ONLY '
*
…

*-- OUVERTURE DU CURSEUR
DEBUT_SQL
  + OPEN C_LIGNE USING
  +   :WN_CARJ ,
  +   :WN_CASY
FIN_SQL

…
```

TRES IMPORTANT : il est très important de noter que la création de requête SQL par concaténation est particulièrement vulnérable aux attaques par injection SQL, si certains éléments de la requête proviennent de données saisies par des utilisateurs, ou proviennent d'applications tierces sur lesquelles on n'a aucun contrôle. La technique de requête paramétrée présentée ci-dessus permet de se prémunir contre ce type de risque. Pour de plus amples informations, prière de se reporter au chapitre dédié à ce sujet, dans le présent document.

10.8 Curseur SQL en mise à jour

ATTENTION : La technique de mise à jour et de suppression présentée dans ce chapitre est quelque peu anecdotique car elle souffre de certaines limitations. Il existe en effet plusieurs types de requêtes pour lesquelles la technique présentée ici ne fonctionne pas, ces requêtes sont les suivantes :

- Les requêtes de jointure,
- Les requêtes contenant les clauses VALUES, ORDER BY, GROUP BY, HAVING ou DISTINCT
- Les requêtes utilisant les modes d'ouverture FOR READ ONLY ou FOR FETCH ONLY

Des solutions de contournement à ces limitations sont indiquées à la fin de ce chapitre.

La documentation d'Hardis contient un exemple de programme effectuant une mise à jour positionnée de ligne à l'intérieur d'un curseur SQL. Nous allons le reprendre et l'enrichir pour effectuer également des suppressions positionnées de lignes, en plus des mises à jour. La technique présentée ici est strictement identique en Adelia et en RPG.

La mise à jour de colonnes à l'intérieur d'un curseur se fait en utilisant l'ordre UPDATE accompagné de la clause WHERE CURRENT OF. On parle de « mise à jour positionnée ».

La suppression de ligne à l'intérieur d'un curseur se fait en utilisant l'ordre DELETE FROM accompagné de la clause WHERE CURRENT OF. On parle de « suppression positionnée ».

L'exemple de code fourni par HARDIS a été remanié ci-dessous pour tenir compte des recommandations déjà énoncées dans les chapitres précédents, qui sont :

- Définir une boucle TANT_QUE avec une condition plus large, comme indiqué dans l'exemple de la page suivante.
- effectuer le FETCH sur une DS (structure hôte) regroupant les zones indiquées dans l'exemple (pour améliorer les performances)
- placer le FETCH dans une procédure, pour améliorer la lisibilité et la maintenance ultérieure
- ajouter le paramètre d'exécution COMMIT = *NONE en début de programme, sinon l'UPDATE ne pourra pas fonctionner (ce problème n'est pas indiqué dans l'exemple d'Hardis). Une autre solution aurait été d'ajouter la clause WITH NC dans la déclaration de la requête elle-même.
- Ajout de la clause FOR UPDATE dans la définition de la requête. Cette clause est facultative, mais le fait de la préciser constitue une bonne pratique, car elle permet à DB2 d'optimiser la gestion des verrouillages pour les lignes concernés par l'opération de mise à jour. Attention : certains auteurs d'ouvrages et d'articles consacrés à DB2 contestent l'intérêt d'une déclaration explicite du mode UPDATE, et semblent préférer le mode d'ouverture dit « ambigü », notamment pour la raison que ce mode permet dans certains cas de limiter le nombre de lignes verrouillées par le curseur.. La clause FOR UPDATE peut être associée à d'autres paramètres (cf complément d'infos en fin de chapitre).
- Transfert de la mise à jour de colonne dans la procédure MAJCUR pour une meilleure

lisibilité du programme. La mise à jour ne sera déclenchée que pour les lignes dont la colonne PCSUP contient une valeur différente de « S ».

- Ajout d'une procédure de suppression (SUPCUR) présentant la méthode permettant d'effectuer une suppression de ligne à l'intérieur du curseur. Cette suppression ne sera déclenchée que pour les lignes dont la colonne PCSUP contient « S ».

Soit un programme batch de calcul d'augmentation des salaires au sein du fichier du personnel (exemple HARDIS remanié) :

```
RECEVOIR WCOSER PTOAUG
*
WTOAUG = PTOAUG
*
DS DSFETCH Z_CODE_SOCIETE , Z_CODE_MATRICUL , Z_MONTANT_SALAI , Z_COD_SUP
*
*-- Désactivation du mode COMMIT (voir chapitre suivant pour + d'infos)
DEBUT_SQL
+ SET OPTION COMMIT = *NONE
FIN_SQL
*
DEBUT_SQL
+ DECLARE EMPLOYES CURSOR FOR
+ SELECT PCOSTE, PCOMAT, PMTSAL, PCSUP
+ FROM HP£PERP
+ WHERE PCOSER = :WCOSER
+ FOR UPDATE
FIN_SQL
*
DEBUT_SQL
+ OPEN EMPLOYES
FIN_SQL
*
*-- Première lecture du curseur
TRAITER_PROC LECCUR
*
TANT_QUE *SQLCODE >= 0 ET *SQLCODE <> 100
  SI Z_COD_SUP = 'S'
     *-- Suppression physique des lignes invalidées logiquement
     TRAITER_PROC SUPCUR
  SINON
     *-- Mise à jour des salaires pour les lignes valides
     Z_MONTANT_SALAI = Z_MONTANT_SALAI + Z_MONTANT_SALAI * (WTOAUG / 100)
     TRAITER_PROC MAJCUR
  FIN
  *-- Lecture suivante du curseur
  TRAITER_PROC LECCUR
REFAIRE
*
DEBUT_SQL
+ CLOSE EMPLOYES
FIN_SQL
*
TERMINER
**************************
DEBUT_PROCEDURE LECCUR
```

```
*************************
  DEBUT_SQL
    + FETCH EMPLOYES                    : lecture ou relecture
    + INTO :DSFETCH
  FIN_SQL
  *
FIN_PROCEDURE
*************************
DEBUT_PROCEDURE SUPCUR
*************************
  *-- Suppression d'une ligne du curseur
  DEBUT_SQL
    + DELETE FROM HP£PERP
    + WHERE CURRENT OF EMPLOYES
  FIN_SQL
  *
FIN_PROCEDURE
*************************
DEBUT_PROCEDURE MAJCUR
*************************
  *-- Mise à jour d'une colonne du curseur
  DEBUT_SQL
    + UPDATE HP£PERP
    + SET PMTSAL = :Z_MONTANT_SALAI
    + WHERE CURRENT OF EMPLOYES
  FIN_SQL
  *
FIN_PROCEDURE
```

Informations complémentaires :

1 - comme indiqué en début de chapitre, on aurait pu omettre la déclaration du mode COMMIT en *NONE dans le SET OPTION, à condition d'ajouter la clause WITH NC dans la déclaration de la requête, de la façon suivante :

```
DEBUT_SQL
+ DECLARE EMPLOYES CURSOR FOR
+ SELECT PCOSTE, PCOMAT, PMTSAL
+ FROM HP£PERP
+ WHERE PCOSER = :WCOSER
+ FOR UPDATE
+ WITH NC    -- voir les chapitres suivants pour plus de précisions
FIN_SQL
```

On recommandera d'utiliser de préférence la clause SET OPTION, car cela évite d'avoir à coder « en dur » sur chaque requête le niveau d'isolement souhaité (cf. chapitre suivant pour une description détaillée des niveaux d'isolement).

2 - on aurait pu déclarer explicitement la(les) colonne(s) en mise à jour, de la façon suivante :

```
DEBUT_SQL
+ DECLARE EMPLOYES CURSOR FOR
+ SELECT PCOSTE, PCOMAT, PMTSAL
+ FROM HP£PERP
+ WHERE PCOSER = :WCOSER
+ FOR UPDATE OF PMTSAL
+ WITH NC
FIN_SQL
```

Si on ne précise pas explicitement la liste des colonnes concernées par la mise à jour, alors toutes les colonnes de la table sont accessibles en mise à jour. Si on précise explicitement la (ou les) colonne(s) en mise à jour, alors il sera impossible de modifier le contenu d'autres colonnes que celle(s) précisée(s) dans la déclaration de la requête. Dans l'exemple ci-dessus, le curseur est ouvert en mise à jour exclusivement pour la colonne PMTSAL.
La réduction du nombre de colonnes en mise à jour par l'intermédiaire de la clause FOR UPDATE OF peut permettre à SQL d'optimiser le traitement de la requête. Cette technique offre également une sécurité intéressante au niveau des processus de mise à jour de la base de données et peut entrer en ligne de compte dans la définition de normes et standards de développement.

3 - Si le curseur sur lequel on souhaite effectuer des mises à jour est concerné par l'un des critères d'exclusion indiqués en début de chapitre, on peut contourner le problème de 2 manières :
- on peut remplacer les 2 requêtes de suppression et de mise à jour par 2 requêtes dynamiques non spécifiquement liées au curseur. Ces requêtes effectueront un UPDATE ou un DELETE en utilisant le principe des requêtes dynamiques, au travers des ordres PREPARE et EXECUTE. Le mot clé USING permettra de passer en paramètre à ces requêtes l'identifiant de la ligne à mettre à jour ou à supprimer.
- on peut remplacer les 2 requêtes de suppression et de mise à jour par des instructions Adelia/RPG natives, ce qui permet de surcroît de s'appuyer sur les mécanismes standards de gestion de verrouillage d'Adelia et du RPG.

4 - On notera que la gestion des verrouillages à l'intérieur de curseurs SQL est plus complexe à gérer en SQL qu'en langage HLL (Adelia ou RPG). De plus les curseurs en mise à jour présentent de moins bonnes performances que les curseurs en lecture seule. Donc avant d'envisager l'utilisation d'un curseur en mise à jour le développeur doit se demander si l'utilisation de ce type de curseur présente un réel avantage par rapport à un traitement en langage HLL, pour le cas qu'il doit traiter. Plusieurs cas de figure peuvent être identifiés :

1. Si la requête considérée est concernée par l'une des restrictions vues précédemment, la question ne se pose pas, le développeur devra recourir à un langage HLL.
2. Si l'ensemble des lignes traitées par le curseur doit faire l'objet d'une mise à jour (comme dans l'exemple de la page précédente), et que ce curseur peut facilement être écrit en langage HLL (Adelia ou RPG), il est certainement plus pertinent d'écrire l'intégralité de la boucle de mise à jour en langage HLL.
3. Si une partie seulement des lignes traitées par le curseur est concernée par une mise à jour ou une suppression (par exemple moins de 60% des lignes), alors on peut envisager de conserver le curseur SQL en lecture seule, et de réécrire les procédures de mise à jour et de suppression en langage HLL ce qui permet de bénéficier des avantages propres à chaque technique d'accès BD. Une autre alternative pourrait consister à stocker le jeu de données retournées par le curseur dans un fichier de travail, puis à relire ce fichier de travail pour procéder aux mises à jour et suppression requises.

La gestion des verrouillages dans les curseurs est abordée de manière très détaillée dans le chapitre suivant.

10.9 Gestion des NULL dans les curseurs

Certaines colonnes d'une table peuvent contenir des valeurs NULL, si la définition de la table l'autorise.

La prise en compte des NULL dans un curseur SQL (ou dans une requête encapsulée dans un programme RPG) nécessite de mettre en œuvre une mécanique un peu particulière.

Dans un curseur dans lequel aucune valeur NULL n'est possible, l'écriture du FETCH se présente de la façon suivante :

```
FETCH curseur INTO :vh1 , :vh2 , :vh3
```

Dans le cas où la variable hôte *vh2* serait susceptible de contenir des valeurs nulles, il est nécessaire d'utiliser la déclaration suivante :

```
FETCH curseur INTO :vh1 , :vh2 :vh2ind , :vh3
```

La variable hôte *vh2ind* est une variable numérique servant d'indicateur pour déterminer le contenu de la variable hôte *vh2*. La détermination du contenu de *vh2* se fait très simplement au moyen d'un test du type :

```
Si vh2ind < 0
   Alors vh2 est NULL
   Sinon vh2 n'est pas NULL
FinSi
```

Dans les faits, *vh2ind* est égal à –1 si *vh2* est NULL, et égal à 0 dans le cas contraire. Mais par convention, on recommande d'utiliser le test « SI indicateur < 0 » comme dans l'exemple ci-dessus.

On retrouve la même problématique dans les requêtes SQL exécutées sans curseur. Par exemple, si dans une requête de type SELECT la colonne vh2 est susceptible de contenir des NULL, alors la requête devra être écrite de la façon suivante :

```
SELECT col1, col2 INTO :vh1, :vh2 :vh2ind FROM <table> WHERE <condition>
```

On pourrait également écrire ceci, qui est strictement equivalent, mais plus verbeux :

```
SELECT col1, col2 INTO :vh1, :vh2 INDICATOR :vh2ind FROM <table>
  WHERE <condition>
```

POINT IMPORTANT : la variable *vh2ind* doit être définie, en Adelia comme en RPG, dans un type compatible avec le type SMALLINT de DB2. On recommandera donc de la créer de type INTEGER et de longueur 5.

REMARQUE : on peut contourner le problème des colonnes NULL en utilisant l'une des fonctions IFNULL, VALUE ou COALESCE, qui sont sensiblement équivalentes. Par exemple, dans la requête ci-dessous, si le contenu de la colonne col2 est NULL, alors la fonction IFNULL renvoie la valeur zéro :

```
SELECT col1, IFNULL(col2, 0) INTO :vh1, :vh2 FROM <table> WHERE <condition>
```

10.10 EXECUTE IMMEDIATE

Il est possible d'exécuter une requête dynamique au moyen de l'ordre SQL « EXECUTE IMMEDIATE ».

On ne peut pas utiliser cette technique sur des requêtes de type SELECT, en revanche on peut l'utiliser sur des requêtes de type CREATE, DROP, INSERT, DELETE ou UPDATE. Cette technique est également très pratique pour la création de table temporaire au moyen de l'instruction « DECLARE GLOBAL TEMPORARY TABLE ».

Quand une requête est exécutée par EXECUTE IMMEDIATE, la chaîne spécifiée est préalablement « parsée » et vérifiée par l'analyseur syntaxique SQL. Si aucune anomalie n'est rencontrée, SQL va rechercher le chemin d'accès le mieux adapté pour la requête considérée.

Premier exemple : la requête ci-dessous consiste à copier dans la table W_PF1 le contenu de la table temporaire W_PF2 :

```
VC_REQ = *BLANK
VC_REQ = 'INSERT INTO ' // W_PF1 /// ' SELECT * FROM QTEMP/' /// W_PF2
*
DEBUT_SQL
  + EXECUTE IMMEDIATE :VC_REQ
FIN_SQL
*
SI *SQLCODE = *NORMAL
   OK…
SINON
   KO…
FIN
```

Second exemple : l'exemple ci-dessous consiste à supprimer puis à recréer la table temporaire MSGTMP :

```
*-- Suppression de la table temporaire
W_REQUETE = *BLANK
W_REQUETE = 'DROP TABLE QTEMP/MSGTMP
*
DEBUT_SQL
  + EXECUTE IMMEDIATE :W_REQUETE
FIN_SQL
*
*-- Recréation de la table temporaire
W_REQUETE = *BLANK
W_REQUETE = 'DECLARE GLOBAL TEMPORARY TABLE MSGTMP (VAL CHAR (30) NOT NULL
               WITH DEFAULT)'
*
DEBUT_SQL
  + EXECUTE IMMEDIATE :W_REQUETE
FIN_SQL
```

Avertissement : toute requête SQL exécutée par l'intermédiaire de EXECUTE IMMEDIATE passe par une phase de parsing et d'analyse, avant d'être exécutée par SQL. Cette technique n'offre donc pas un niveau de performances optimal. On la réservera de préférence à des cas de requêtes exécutées une seule fois au sein d'un même programme. Dans le cas de requêtes exécutées plusieurs fois au sein d'un même programme, on utilisera de préférence les ordres SQL PREPARE et EXECUTE présentés au chapitre suivant. Cette autre technique permet à SQL de n'analyser qu'une seule fois une requête destinée à être exécutée plusieurs fois.

La seconde partie de l'annexe présente un exemple de programme SQLRPGLE contenant une requête très longue exécutée selon le principe du EXECUTE IMMEDIATE. Elle était en effet trop longue pour pouvoir être parsée par SQL via l'odre PREPARE EXECUTE (cf. commentaire complémentaire à la fin du chapitre suivant).

10.11 PREPARE … EXECUTE …

On a vu dans le chapitre précédent que l'on pouvait exécuter des requêtes unitaires de manière dynamique avec l'instruction EXECUTE IMMEDIATE. Dans le premier exemple du chapitre précédent, qui est rappelé ci-dessous, on constitue la requête par une concaténation de variables :

Exemple de requête consistant à copier une table temporaire dans une autre :

```
RECEVOIR W_PF1 W_PF2
*
VC_REQ = *BLANK
VC_REQ = 'INSERT INTO ' // W_PF1 /// ' SELECT * FROM QTEMP/' /// W_PF2
*
DEBUT_SQL
  + EXECUTE IMMEDIATE :VC_REQ
FIN_SQL
*
SI *SQLCODE = *NORMAL
   OK…
SINON
   KO…
FIN
```

Même requête exécutée au moyen des ordres PREPARE et EXECUTE :

```
RECEVOIR W_PF1 W_PF2
*
VC_REQ = *BLANK
VC_REQ = 'INSERT INTO ' // W_PF1 /// ' SELECT * FROM QTEMP/' /// W_PF2
*
*-- Préparation de la requête REQ1
DEBUT_SQL
  + PREPARE REQ1 FROM :VC_REQ
FIN_SQL
*
*-- Exécution de la requête REQ1
DEBUT_SQL
  + EXECUTE REQ1
FIN_SQL
*
SI *SQLCODE = *NORMAL
   OK…
SINON
   KO…
FIN
```

Comme indiqué au chapitre précédent, l'utilisation des ordres PREPARE et EXECUTE devra être préféré à EXECUTE IMMEDIATE, à chaque fois qu'une même requête doit être exécutée plusieurs fois.

De plus, les ordres PREPARE et EXECUTE offrent un énorme avantage qui est de pouvoir être associé à l'ordre USING. Cet ordre permet de transmettre à la requête des variables hôtes, par l'intermédiaire de points d'interrogation qui sont appelés des paramètres marqueurs (en anglais :

« parameter markers ») dans le jargon SQL.

Exemple de requête constituée contenant des variables hôtes transmises par substitution de ? :

```
RECEVOIR W_PF1 W_PF2
*
VC_REQ = *BLANK
VC_REQ = 'INSERT INTO MATABLE SELECT * FROM QTEMP/TOTO WHERE COL1 = ? AND COL2
= ?' ;
*
*-- Préparation de la requête au moyen de l'instruction SQL PREPARE
DEBUT_SQL
   + PREPARE REQ1 FROM :VC_REQ
FIN_SQL
*
*-- Exécution de la requête au moyen de l'instruction SQL EXECUTE
*     avec paramètres marqueurs au moyen de USING
DEBUT_SQL
   + EXECUTE REQ1 USING
   + :W_PF1 ,
   + :W_PF2
FIN_SQL
*
SI *SQLCODE = *NORMAL
   OK…
SINON
   KO…
FIN
```

La technique présentée ci-dessus simplifie l'écriture des requêtes en épargnant au développeur la phase - souvent délicate - de concaténation des différents éléments composant une requête. Elle a également le mérite de préserver des risques d'attaque par injection SQL présentés au chapitre suivant.

Second exemple de requête avec paramètres marqueurs :

```
* Préparation de la requête dynamique (4 param. de substitution)
WREQUETE = 'Insert into FICHIER 'VAL1, VAL2, VAL3, VAL4 '
WREQUETE = WREQUETE /// ' values(?,?,?,?) with nc'
*
DEBUT_SQL
  + PREPARE REQ2 FROM :WREQUETE
FIN_SQL
*
* Préparation des variables de substitution
WK_VAL1    = 'E'
WK_VAL2    = 'A'
WK_VAL3    = 10
WK_VAL4    = 150
*
DEBUT_SQL
  + EXECUTE REQ2 USING
  + :WK_VAL1 ,
```

```
  + :WK_VAL2 ,
  + :WK_VAL3 ,
  + :WK_VAL4
FIN_SQL
```

POINT IMPORTANT : il ne faut surtout pas utiliser de paramètres marqueurs sur les noms de fichiers définis dans la clause FROM, ni sur des noms de colonnes définis dans la clause SELECT.

Exemples de requêtes que DB2 n'est pas en mesure de traiter :

```
SELECT * FROM ? WHERE col1 = 'x' ;

SELECT col1, col2, ? FROM table WHERE col1 = 'x' ;
```

10.12 Protection contre les attaques par injection de code

Les développeurs IBMi sont généralement peu familiarisés avec les risques d'attaques par injection SQL, qui sont monnaie courante dans le développement web. Avec l'ouverture des applications IBMi au web, mais aussi avec l'intégration de services webs et de données provenant de systèmes hétérogènes, il est important de sensibiliser les développeurs IBMi avec les bonnes pratiques de codage SQL permettant d'éviter certains types d'attaques.

Exemple de requête classique :

```
SELECT COUNT(*) FROM TIERSB WHERE TIBN8PK = 'CALBERSON LYON'
COUNT ( * )
          25
```

La clause WHERE de la requête ci-dessus peut être constituée de manière dynamique, par une concaténation de chaîne telle que celle-ci-desous :

```
W_REQUETE = 'SELECT TIBCARJ FROM TIERSB WHERE TIBN8PK = ''' ///
            W_CONDITION /// ''''
```

La requête ci-dessus est particulièrement vulnérable aux attaques dites « par injection SQL ». Démonstration ci-dessous :

- 1er exemple d'attaque par injection SQL :

Si W_CONDITION contient la chaîne suivante : CALBERSON LYON' or "="
Alors la concaténation de la chaîne donnera le résultat suivant :

```
SELECT COUNT(*) FROM TIERSB
  WHERE TIBN8PK = 'CALBERSON LYON' or ''=''
COUNT ( * )
     19.476
```

- 2ème exemple produisant le même effet :

Si W_CONDITION contient la chaîne suivante : CALBERSON LYON' or '0' = '0'
Alors la concaténation de la chaîne donnera le résultat suivant :

```
SELECT COUNT(*) FROM TIERSB
  WHERE TIBN8PK = 'CALBERSON LYON' or '0' = '0'
COUNT ( * )
     19.476
```

- 3ème exemple produisant le même effet :

Si W_CONDITION contient la chaîne suivante : ' or 0 --
Alors la concaténation de la chaîne donnera le résultat suivant :

```
SELECT COUNT(*) FROM TIERSB
  WHERE TIBN8PK = '' or 0 --
COUNT ( * )
     19.476
```

N.B. : les 2 tirets correspondent au début d'une chaîne de commentaire pour SQL, cela peut permettre de court-circuiter une bonne partie d'une requête SQL. Par exemple, si la requête était formée de la façon suivante :

```
W_REQUETE = 'SELECT TIBCARJ FROM TIERSB WHERE TIBN8PK = ''' ///
             W_CONDITION /// ''''
W_REQUETE = W_REQUETE /// ' ORDER BY TIBCARJ'
```

On obtiendrait à l'exécution la chaîne ci-dessous, parfaitement acceptée par l'analyseur SQL, qui va ignorer tout ce qui suit les 2 tirets :

```
SELECT TIBCARJ FROM TIERSB WHERE TIBN8PK = '' or 1=1 --' ORDER BY TIBCARJ
COUNT ( * )
     19.476
```

Recommandations :
Pour éviter ce genre de problème, on recommandera au développeur d'utiliser chaque fois que c'est possible, et surtout si les valeurs utilisées dans la clause WHERE sont saisies par les utilisateurs, la forme d'écriture exploitant les caractères de substitution (soit le ? utilisé conjointement avec les clauses PREPARE, EXECUTE et USING). Cette technique d'écriture est présentée au chapitre précédent.

On notera également que les formes d'écriture basées sur des requêtes statiques ne présentent pas, à prori, ce genre de vulnérabilité.

Une solution préventive peut également être envisagée, qui consisterait à appliquer sur la variable W_CONDITION, avant de la concaténer à la variable W_REQUETE, une suppression de tous les caractères susceptibles de vulnérabiliser la requête. Les caractères qui apparaissent comme les plus dangereux et qu'il conviendrait d'éliminer sont les suivants :

=	Egal (voir exemples ci-dessus)
?	Utilisé pour les paramètres marqueurs
-	Utilisé pour la mise en commentaire
|	Caractère de concaténation SQL (équivalent à CONCAT)
!	Caractère équivalent à CONCAT mais spécifique à SQL/400

Ce nettoyage de chaîne pourrait être réalisé avec une règle de gestion Adelia, elle serait appliquée

systématiquement à toute chaîne provenant de l'extérieur, qu'il s'agisse d'une saisie utilisateur, ou de donnée provenant d'une base de données, de services webs ou de traitements EDI.

10.13 Fonctions scalaires sans requêtes

Il est possible d'utiliser les fonctions scalaires du SQL au sein d'un programme Adelia, sans pour autant exécuter une requête SQL nécessitant un accès base de données.

Les meilleures fonctions candidates à ce type d'utilisation sont les fonctions de manipulation des chaînes de caractères (TRANSLATE, REPLACE, etc…), les fonctions de manipulation des dates (DATE, DAY, DAYOFMONTH, etc…), ou encore les fonctions mathématiques (SIN, COS, etc…).

Exemples d'utilisation de fonctions scalaires SQL en Adelia (idem en RPG) :

```
*-- Suppression caractères accentués et forçage en majuscule
DEBUT_SQL
 + SET :ZONE1 = UPPER(TRANSLATE(:ZONE1,
 +            '            eeeeaaaiiuuuoocEEEEAAAIIUUUOOC',
 +            '¼½¾²³®@©æÆ°µéèêëàâäïîùûüôöçÉÈÊËÀÂÄÏÎÙÛÜÔÖÇ')),
FIN_SQL
*
*--  Remplacement de "&" par "ET" dans ZONE2 et recadrage à gauche
DEBUT_SQL
 + SET :ZONE2 = LTRIM(REPLACE(:ZONE2 , '&' , 'ET')
FIN_SQL
```

On peut utiliser l'expression VALUES au lieu de SET. Par exemple, la requête suivante :

```
DEBUT_SQL
 + SET :ZONE2 = LTRIM(REPLACE(:ZONE2 , '&' , 'ET')
FIN_SQL
```

est strictement équivalente à la requête suivante :

```
DEBUT_SQL
 + VALUES (LTRIM(REPLACE(:ZONE2 , '&' , 'ET')) INTO :ZONE2
FIN_SQL
```

De même, la requête suivante :

```
DEBUT_SQL
   + SET :W_COUNT = (SELECT COUNT(*) FROM DBTEST)
FIN_SQL
```

est strictement équivalente à la requête suivante :

```
DEBUT_SQL
   + VALUES (SELECT COUNT(*) FROM DBTEST) INTO :W_COUNT
FIN_SQL
```

Dans l'exemple ci-dessous, la fonction TRANSLATE a été utilisée pour supprimer dans la variable W£CORI certains caractères parasites susceptibles de provoquer une erreur programme :

```
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'01')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'02')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'03')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'04')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'10')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'11')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'12')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'13')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'14')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'15')
FIN_SQL
DEBUT_SQL
   +SET :W£CORI = TRANSLATE(:W£CORI, ' ', X'1D')
FIN_SQL
```

11. Les Jointures

Ce chapitre présente les différents types de jointures SQL.

Il s'appuie majoritairement sur un article du site foothing.net :

On dénombre 5 grands types de jointures, qui sont les suivants :

1. CROSS JOIN : effectue un produit cartésien entre le contenu de deux tables.
2. INNER JOIN : permet de faire une jointure entre deux tables et de ramener les enregistrements de la première table qui ont une correspondance dans la seconde table.
3. LEFT OUTER JOIN : cette jointure entre deux tables retourne tous les enregistrements ramenés par un INNER JOIN plus chaque enregistrement de la table 1 qui n'a pas de correspondance dans la table 2.
4. RIGHT OUTER JOIN : Cette technique de jointure retourne tous les enregistrements ramenés par un INNER JOIN plus chaque enregistrement de la table 2 qui n'a pas de correspondance dans la table 1.
5. EXCEPTION JOIN : retourne uniquement les enregistrements de la table 1 qui n'ont pas de correspondance dans la table 2.

Exemple utilisé

Considérons les quatre tables EMPLOYE, PROJET, DEPARTEMENT et ACTIVITES. Elles nous serviront d'exemple pour chaque type de jointure.

Règles de gestion :

- Chaque employé figure dans la table EMPLOYE.
- Un employé responsable d'un projet figure dans la table PROJET.
- Un employé fait partie d'un département.
- La liste des départements se trouve dans la table DEPARTEMENT.
- La table ACTIVITES contient une liste des activités assurées par le CE de l'entreprise.

Table des employés

```
EMPLOYE,
 EMPNO, numéro de l'employé
 LASTNAME, nom de l'employé
 WORKDEPT, département ou travaille l'employé
```

EMPNO	LASTNAME	WORKDEPT
000020	THOMPSON	000001
000060	STERN	000002
000100	SPENSER	000003
000170	YOSHIMURA	000002

000180	SCOUTTEN	000002
000190	WALKER	000002
000250	SMITH	000004
000280	SCHNEIDER	000005
000300	SMITH	000005
000310	SETRIGHT	000005

Table des employés responsable d'un projet

```
PROJET,
 RESPEMP, numéro de l'employé responsable du projet
 PROJNO, numéro du projet
```

RESPEMP	PROJNO
000020	PL2100
000060	MA2110
000100	OP2010
000250	AD3112

Table des départements

```
DEPARTEMENT,
 DEPTNO, numéro de département
 DEPTNAME, Nom du département
```

DEPTNO	DEPTNAME
000001	PLANNING
000002	MANUFACTURING SYSTEMS
000003	SOFTWARE SUPPORT
000004	ADMINISTRATION SYSTEMS
000005	OPERATIONS

Table des activités du CE

```
ACTIVITES,
 ACTINAME, Nom activités
```

ACTINAME
TENNIS
CINEMA
COURS DE LANGUE

CROSS JOIN

Le **Cross join** effectue un produit cartésien entre le contenu de deux tables.

Les deux requêtes suivantes produisent le même résultat

```
SELECT * FROM EMPLOYE CROSS JOIN ACTIVITES

SELECT * FROM EMPLOYE, ACTIVITES
```

Résultat

EMPNO	LASTNAME	WORKDEPT	ACTINAME
000020	THOMPSON	000001	TENNIS

000020	THOMPSON	000001	CINEMA
000020	THOMPSON	000001	COURS DE LANGUE
000060	STERN	000002	TENNIS
000060	STERN	000002	CINEMA
000060	STERN	000002	COURS DE LANGUE
000100	SPENSER	000003	TENNIS
000100	SPENSER	000003	CINEMA
000100	SPENSER	000003	COURS DE LANGUE
000170	YOSHIMURA	000002	TENNIS
000170	YOSHIMURA	000002	CINEMA
000170	YOSHIMURA	000002	COURS DE LANGUE
000180	SCOUTTEN	000002	TENNIS
000180	SCOUTTEN	000002	CINEMA
000180	SCOUTTEN	000002	COURS DE LANGUE
000190	WALKER	000002	TENNIS
000190	WALKER	000002	CINEMA
000190	WALKER	000002	COURS DE LANGUE
000250	SMITH	000004	TENNIS
000250	SMITH	000004	CINEMA
000250	SMITH	000004	COURS DE LANGUE
000280	SCHNEIDER	000005	TENNIS
000280	SCHNEIDER	000005	CINEMA
000280	SCHNEIDER	000005	COURS DE LANGUE
000300	SMITH	000005	TENNIS
000300	SMITH	000005	CINEMA
000300	SMITH	000005	COURS DE LANGUE
000310	SETRIGHT	000005	TENNIS
000310	SETRIGHT	000005	CINEMA
000310	SETRIGHT	000005	COURS DE LANGUE

INNER JOIN

Le **INNER JOIN** permet de faire une jointure entre deux tables et de ramener les enregistrements de la première table qui ont une correspondance dans la seconde table.

Par exemple, ramener le numéro d'employé, le nom d'employé et le numéro du projet dont les employés sont responsables.

Les deux requêtes suivantes produisent le même résultat.

```
SELECT EMPNO, LASTNAME, PROJNO
   FROM EMPLOYE INNER JOIN PROJET
         ON EMPNO = RESPEMP

SELECT EMPNO, LASTNAME, PROJNO
   FROM EMPLOYE, PROJET
   WHERE EMPNO = RESPEMP
```

Résultat

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000250	SMITH	AD3112

LEFT OUTER JOIN

Le **LEFT OUTER JOIN** entre deux tables retourne tous les enregistrements ramenés par un **INNER JOIN** plus chaque enregistrement de la table 1 qui n'a pas de correspondance dans la table 2.

Le résultat de la requête suivante ramène tous les employés, même ceux qui n'ont pas de numéro de projet. Dans ce cas, une valeur nulle est retournée à la place du numéro de projet.

```
SELECT EMPNO, LASTNAME, PROJNO
   FROM EMPLOYE LEFT OUTER JOIN PROJET
         ON EMPNO = RESPEMP
```

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-

Si l'on souhaite se débarasser des Null dans le jeu de données résultant, on peut recourir à la fonction IFNULL, comme dans l'exemple ci-dessous :

```
SELECT EMPNO, LASTNAME, IFNULL(PROJNO, '') AS PROJNO
   FROM EMPLOYE
   LEFT OUTER JOIN PROJET
         ON EMPNO = RESPEMP
```

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	
000180	SCOUTTEN	
000190	WALKER	
000250	SMITH	AD3112
000280	SCHNEIDER	
000300	SMITH	

000310	SETRIGHT	

RIGHT OUTER JOIN

Le **RIGHT OUTER JOIN** entre deux tables retourne tous les enregistrements ramenés par un **INNER JOIN** plus chaque enregistrement de la table 2 qui n'a pas de correspondance dans la table 1.

Le résultat de la requête suivante ramène tous les employés, même ceux qui n'ont pas de numéro de projet. Dans ce cas, une valeur nulle est retournée à la place du numéro de projet.

Le résultat est le même que celle du LEFT OUTER JOIN.

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM PROJET RIGHT OUTER JOIN EMPLOYE
        ON EMPNO = RESPEMP
```

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000060	STERN	MA2110
000100	SPENSER	OP2010
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000250	SMITH	AD3112
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-

EXCEPTION JOIN

Un **EXCEPTION JOIN** retourne uniquement les enregistrements de la table 1 qui n'on pas de correspondance dans la table 2.

En utilisant le même exemple, la requête suivante ramène uniquement la liste des employés qui ne sont responsable d'aucun projet.

```
SELECT EMPNO, LASTNAME, PROJNO
  FROM EMPLOYE EXCEPTION JOIN PROJET
        ON EMPNO = RESPEMP
```

EMPNO	LASTNAME	PROJNO
000170	YOSHIMURA	-
000180	SCOUTTEN	-
000190	WALKER	-
000280	SCHNEIDER	-
000300	SMITH	-
000310	SETRIGHT	-

On peut écrire la même requête en utilisant le prédicat NOT EXISTS comme dans l'exemple suivant. La seule différence est que cette requête ne peut retourner de valeur de la table PROJET.

```
SELECT EMPNO, LASTNAME
  FROM EMPLOYE
  WHERE NOT EXISTS
        (SELECT * FROM PROJET
           WHERE EMPNO = RESPEMP)
```

MULTIPLES JOINS DANS UNE SEULE REQUETE

Il est bien sur possible de faire plusieurs jointures dans une même requête comme l'indique l'exemple suivant.

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM EMPLOYE INNER JOIN DEPARTEMENT
         ON WORKDEPT = DEPTNO
       LEFT OUTER JOIN PROJET
         ON EMPNO = RESPEMP
```

EMPNO	LASTNAME	DEPTNAME	PROJNO
000020	THOMPSON	PLANNING	PL2100
000060	STERN	MANUFACTURING SYSTEMS	MA2110
000100	SPENSER	SOFTWARE SUPPORT	OP2010
000170	YOSHIMURA	MANUFACTURING SYSTEMS	-
000180	SCOUTTEN	MANUFACTURING SYSTEMS	-
000190	WALKER	MANUFACTURING SYSTEMS	-
000250	SMITH	ADMINISTRATION SYSTEMS	AD3112
000280	SCHNEIDER	OPERATIONS	-
000300	SMITH	OPERATIONS	-
000310	SETRIGHT	OPERATIONS	-

Exemple concret

CSTEQH est le code société.
CAGCQH est le code agence.
La requête suivante sélectionne tous les couples société/agence du fichier STKSFMW et les valeurs de stock correspondantes dans STKSFM.
Si les valeurs ne correspondent pas dans STKSFM pour un couple société/agence, elles sont à valeur nulle.
Les enregistrements sélectionnés sont insérés dans le fichier STKSFMW1.
Le code de la requête SQL est extrait d'un source ADELIA.

```
DEBUT_SQL
+  INSERT
+  INTO QTEMP/STKSFMW1
+ (CSTEQH, CAGCQH, MOECQH, ACALQH, CARTQH, QTPHQH, CDUPQH,
+  CDUFQH, UPUFQH, PRRFQH, DISPQH, QDISQH, JRUPQH, CPROQH)
+  SELECT
+  A.CSTEQH, A.CAGCQH, B.MOECQH, B.ACALQH,
+  B.CARTQH, B.QTPHQH, B.CDUPQH, B.CDUFQH,
+  B.UPUFQH, B.PRRFQH, B.DISPQH, B.QDISQH,
+  B.JRUPQH, B.CPROQH
+  FROM
+  QTEMP/STKSFMW  AS A
+  LEFT OUTER JOIN
+  STKSFM  AS B
+  ON  A.CSTEQH = B.CSTEQH AND
+  A.CARTQH = B.CARTQH AND A.CAGCQH = B.CAGCQH
+  AND (B.ACALQH * 100 + B.MOECQH ) = A.MAXDAT
+  WHERE B.QTPHQH <> 0 OR B.QDISQH <> 0
FIN_SQL
```

POINT IMPORTANT :

Afin d'améliorer la lisibilité et la maintenabilité des requêtes, il est préférable de TOUJOURS attribuer un alias, à chacune des tables utilisées dans une requête. Par exemple, la requête suivante :

```
SELECT EMPNO, LASTNAME, PROJNO
   FROM EMPLOYE
   INNER JOIN PROJET
         ON EMPNO = RESPEMP
```

... est nettement plus lisible et maintenable si chaque table a un alias la définissant précisément (comme ici avec les codes A et B) :

```
SELECT A.EMPNO, A.LASTNAME, B.PROJNO
   FROM EMPLOYE A
   INNER JOIN PROJET B
         ON A.EMPNO = B.RESPEMP
```

... ou si l'on préfère un code mnémotechnique sur 3 caractères :

```
SELECT EMP.EMPNO, EMP.LASTNAME, PRO.PROJNO
   FROM EMPLOYE EMP
   INNER JOIN PROJET PRO
         ON EMP.EMPNO = PRO.RESPEMP
```

12. ANNEXES

12.1 Listes des pays au format SQL

La table SQL ci-dessous nous servira de table exemple pour tester différentes techniques, notamment dans le chapitre 6.

```
CREATE TABLE MYLIBRARY.LSTPAYS (
  CODFRA CHAR (3 ) NOT NULL WITH DEFAULT,
  CODISO CHAR (2 ) NOT NULL WITH DEFAULT,
  LIBELLE CHAR (50 ) NOT NULL WITH DEFAULT
) ;

-- DELETE FROM MYLIBRARY.LSTPAYS ;
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AFG ', 'AF ', 'AFGHANISTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ZAF ', 'ZA ', 'AFRIQUE DU SUD');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ALA ', 'AX ', 'ALAND, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ALB ', 'AL ', 'ALBANIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DZA ', 'DZ ', 'ALGERIE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DEU ', 'DE ', 'ALLEMAGNE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AND ', 'AD ', 'ANDORRE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AGO ', 'AO ', 'ANGOLA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AIA ', 'AI ', 'ANGUILLA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ATA ', 'AQ ', 'ANTARCTIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ATG ', 'AG ', 'ANTIGUA-ET-BARBUDA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ANT ', 'AN ', 'ANTILLES NEERLANDAISES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SAU ', 'SA ', 'ARABIE SAOUDITE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ARG ', 'AR ', 'ARGENTINE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ARM ', 'AM ', 'ARMENIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ABW ', 'AW ', 'ARUBA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AUS ', 'AU ', 'AUSTRALIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AUT ', 'AT ', 'AUTRICHE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('AZE ', 'AZ ', 'AZERBAIDJAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BHS ', 'BS ', 'BAHAMAS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BHR ', 'BH ', 'BAHREIN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BGD ', 'BD ', 'BANGLADESH');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BRB ', 'BB ', 'BARBADE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BLR ', 'BY ', 'BELARUS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BEL ', 'BE ', 'BELGIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BLZ ', 'BZ ', 'BELIZE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BEN ', 'BJ ', 'BENIN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BMU ', 'BM ', 'BERMUDES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BTN ', 'BT ', 'BHOUTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BOL ', 'BO ', 'BOLIVIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BIH ', 'BA ', 'BOSNIE-HERZEGOVINE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BWA ', 'BW ', 'BOTSWANA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BVT ', 'BV ', 'BOUVET, ILE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BRA ', 'BR ', 'BRESIL ');
```

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BRN ', 'BN ', 'BRUNEI DARUSSALAM ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BGR ', 'BG ', 'BULGARIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BFA ', 'BF ', 'BURKINA FASO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BDI ', 'BI ', 'BURUNDI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CYM ', 'KY ', 'CAIMANES, ILES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KHM ', 'KH ', 'CAMBODGE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CMR ', 'CM ', 'CAMEROUN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CAN ', 'CA ', 'CANADA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CPV ', 'CV ', 'CAP-VERT');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CAF ', 'CF ', 'CENTRAFRICAINE, REPUBLIQUE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CHL ', 'CL ', 'CHILI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CHN ', 'CN ', 'CHINE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CXR ', 'CX ', 'CHRISTMAS, ILE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CYP ', 'CY ', 'CHYPRE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CCK ', 'CC ', 'COCOS (KEELING), ILES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('COL ', 'CO ', 'COLOMBIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('COM ', 'KM ', 'COMORES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('COG ', 'CG ', 'CONGO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('COD ', 'CD ',
       'CONGO, LA REPUBLIQUE DEMOCRATIQUE DU ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('COK ', 'CK ', 'COOK, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KOR ', 'KR ', 'COREE, REPUBLIQUE DE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PRK ', 'KP ',
       'COREE, REPUBLIQUE POPULAIRE DEMOCRATIQUE DE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CRI ', 'CR ', 'COSTA RICA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CIV ', 'CI ', 'COTE D''IVOIRE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HRV ', 'HR ', 'CROATIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CUB ', 'CU ', 'CUBA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DNK ', 'DK ', 'DANEMARK');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DJI ', 'DJ ', 'DJIBOUTI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DOM ', 'DO ', 'DOMINICAINE, REPUBLIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('DMA ', 'DM ', 'DOMINIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('EGY ', 'EG ', 'EGYPTE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SLV ', 'SV ', 'EL SALVADOR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ARE ', 'AE ', 'EMIRATS ARABES UNIS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ECU ', 'EC ', 'EQUATEUR ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ERI ', 'ER ', 'ERYTHREE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ESP ', 'ES ', 'ESPAGNE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('EST ', 'EE ', 'ESTONIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('USA ', 'US ', 'Etats-Unis');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ETH ', 'ET ', 'ETHIOPIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FLK ', 'FK ', 'FALKLAND, ILES (MALVINAS) ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FRO ', 'FO ', 'FEROE, ILES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FJI ', 'FJ ', 'FIDJI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FIN ', 'FI ', 'FINLANDE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FRA ', 'FR ', 'FRANCE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GAB ', 'GA ', 'GABON');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GMB ', 'GM ', 'GAMBIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GEO ', 'GE ', 'GEORGIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SGS ', 'GS ',
       'GEORGIE DU SUD ET LES ILES SANDWICH DU SUD');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GHA ', 'GH ', 'GHANA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GIB ', 'GI ', 'GIBRALTAR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GRC ', 'GR ', 'GRECE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GRD ', 'GD ', 'GRENADE');
```

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GRL ', 'GL ', 'GROENLAND');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GLP ', 'GP ', 'GUADELOUPE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GUM ', 'GU ', 'GUAM');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GTM ', 'GT ', 'GUATEMALA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GGY ', 'GG ', 'GUERNESEY');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GIN ', 'GN ', 'GUINEE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GNB ', 'GW ', 'GUINEE BISSAU ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GNQ ', 'GQ ', 'GUINEE EQUATORIALE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GUY ', 'GY ', 'GUYANA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GUF ', 'GF ', 'GUYANE FRANCAISE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HTI ', 'HT ', 'HAITI ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HMD ', 'HM ',
       'HEARD, ILE ET MCDONALD, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HND ', 'HN ', 'HONDURAS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HKG ', 'HK ', 'HONG KONG');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('HUN ', 'HU ', 'HONGRIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IMN ', 'IM ', 'ILE DE MAN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('UMI ', 'UM ',
       'ILES MINEURES ELOIGNEES DES ETATS-UNIS ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VGB ', 'VG ',
       'ILES VIERGES BRITANNIQUES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VIR ', 'VI ',
       'ILES VIERGES DES ETATS-UNIS ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IND ', 'IN ', 'INDE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IDN ', 'ID ', 'INDONESIE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IRN ', 'IR ',
       'IRAN, REPUBLIQUE ISLAMIQUE D'' ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IRQ ', 'IQ ', 'IRAQ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IRL ', 'IE ', 'IRLANDE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ISL ', 'IS ', 'ISLANDE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ISR ', 'IL ', 'ISRAEL ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ITA ', 'IT ', 'ITALIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('JAM ', 'JM ', 'JAMAIQUE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('JPN ', 'JP ', 'JAPON');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('JEY ', 'JE ', 'JERSEY');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('JOR ', 'JO ', 'JORDANIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KAZ ', 'KZ ', 'KAZAKHSTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KEN ', 'KE ', 'KENYA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KGZ ', 'KG ', 'KIRGHIZISTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KIR ', 'KI ', 'KIRIBATI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KWT ', 'KW ', 'KOWEIT ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LAO ', 'LA ',
       'LAOS, REPUBLIQUE DEMOCRATIQUE POPULAIRE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LSO ', 'LS ', 'LESOTHO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LVA ', 'LV ', 'LETTONIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LBN ', 'LB ', 'LIBAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LBR ', 'LR ', 'LIBERIA ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LBY ', 'LY ',
       'LIBYENNE, JAMAHIRIYA ARABE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LIE ', 'LI ', 'LIECHTENSTEIN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LTU ', 'LT ', 'LITUANIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LUX ', 'LU ', 'LUXEMBOURG');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MAC ', 'MO ', 'MACAO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MKD ', 'MK ',
       'MACEDOINE, L''EX-REPUBLIQUE YOUGOSLAVE DE ');
```

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MDG ', 'MG ', 'MADAGASCAR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MYS ', 'MY ', 'MALAISIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MWI ', 'MW ', 'MALAWI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MDV ', 'MV ', 'MALDIVES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MLI ', 'ML ', 'MALI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MLT ', 'MT ', 'MALTE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MNP ', 'MP ',
       'MARIANNES DU NORD, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MAR ', 'MA ', 'MAROC');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MHL ', 'MH ', 'MARSHALL, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MTQ ', 'MQ ', 'MARTINIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MUS ', 'MU ', 'MAURICE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MRT ', 'MR ', 'MAURITANIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MYT ', 'YT ', 'MAYOTTE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MEX ', 'MX ', 'MEXIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('FSM ', 'FM ',
       'MICRONESIE, ETATS FEDERES DE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MDA ', 'MD ', 'MOLDOVA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MCO ', 'MC ', 'MONACO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MNG ', 'MN ', 'MONGOLIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MNE ', 'ME ', 'MONTENEGRO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MSR ', 'MS ', 'MONTSERRAT');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MOZ ', 'MZ ', 'MOZAMBIQUE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MMR ', 'MM ', 'MYANMAR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NAM ', 'NA ', 'NAMIBIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NRU ', 'NR ', 'NAURU');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NPL ', 'NP ', 'NEPAL ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NIC ', 'NI ', 'NICARAGUA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NER ', 'NE ', 'NIGER');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NGA ', 'NG ', 'NIGERIA ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NIU ', 'NU ', 'NIUE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NFK ', 'NF ', 'NORFOLK, ILE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NOR ', 'NO ', 'NORVEGE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NCL ', 'NC ', 'NOUVELLE-CALEDONIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NZL ', 'NZ ', 'NOUVELLE-ZELANDE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('IOT ', 'IO ',
       'OCEAN INDIEN, TERRITOIRE BRITANNIQUE DE L'' ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('OMN ', 'OM ', 'OMAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('UGA ', 'UG ', 'OUGANDA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('UZB ', 'UZ ', 'OUZBEKISTAN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PAK ', 'PK ', 'PAKISTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PLW ', 'PW ', 'PALAOS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PSE ', 'PS ',
       'PALESTINIEN OCCUPE, TERRITOIRE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PAN ', 'PA ', 'PANAMA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PNG ', 'PG ',
       'PAPOUASIE-NOUVELLE-GUINEE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PRY ', 'PY ', 'PARAGUAY');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('NLD ', 'NL ', 'PAYS-BAS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PER ', 'PE ', 'PEROU ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PHL ', 'PH ', 'PHILIPPINES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PCN ', 'PN ', 'PITCAIRN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('POL ', 'PL ', 'POLOGNE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PYF ', 'PF ',
       'POLYNESIE FRANCAISE ');
```

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PRI ', 'PR ', 'PORTO RICO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('PRT ', 'PT ', 'PORTUGAL');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('QAT ', 'QA ', 'QATAR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('REU ', 'RE ', 'REUNION ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ROU ', 'RO ', 'ROUMANIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('GBR ', 'GB ', 'ROYAUME-UNI');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('RUS ', 'RU ',
       'RUSSIE, FEDERATION DE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('RWA ', 'RW ', 'RWANDA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ESH ', 'EH ', 'SAHARA OCCIDENTAL');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('BLM ', 'BL ', 'SAINT-BARTHELEMY');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('KNA ', 'KN ', 'SAINT-KITTS-ET-NEVIS');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SMR ', 'SM ', 'SAINT-MARIN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('MAF ', 'MF ',
       'SAINT-MARTIN (PARTIE FRANCAISE) ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SPM ', 'PM ',
       'SAINT-PIERRE-ET-MIQUELON');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VAT ', 'VA ',
       'SAINT-SIEGE (ETAT DE LA CITE DU VATICAN)');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VCT ', 'VC ',
       'SAINT-VINCENT-ET-LES GRENADINES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SHN ', 'SH ', 'SAINTE-HELENE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LCA ', 'LC ', 'SAINTE-LUCIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SLB ', 'SB ', 'SALOMON, ILES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('WSM ', 'WS ', 'SAMOA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ASM ', 'AS ', 'SAMOA AMERICAINES ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('STP ', 'ST ', 'SAO TOME-ET-PRINCIPE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SEN ', 'SN ', 'SENEGAL ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SRB ', 'RS ', 'SERBIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SYC ', 'SC ', 'SEYCHELLES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SLE ', 'SL ', 'SIERRA LEONE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SGP ', 'SG ', 'SINGAPOUR');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SVK ', 'SK ', 'SLOVAQUIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SVN ', 'SI ', 'SLOVENIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SOM ', 'SO ', 'SOMALIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SDN ', 'SD ', 'SOUDAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('LKA ', 'LK ', 'SRI LANKA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SWE ', 'SE ', 'SUEDE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CHE ', 'CH ', 'SUISSE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SUR ', 'SR ', 'SURINAME');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SJM ', 'SJ ', 'SVALBARD ET ILE JAN MAYEN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SWZ ', 'SZ ', 'SWAZILAND');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('SYR ', 'SY ', 'SYRIENNE, REPUBLIQUE ARABE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TJK ', 'TJ ', 'TADJIKISTAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TWN ', 'TW ', 'TAIWAN');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TZA ', 'TZ ', 'TANZANIE, REPUBLIQUE UNIE DE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TCD ', 'TD ', 'TCHAD');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('CZE ', 'CZ ', 'TCHEQUE, REPUBLIQUE ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ATF ', 'TF ', 'TERRES AUSTRALES FRANCAISES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('THA ', 'TH ', 'THAILANDE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TLS ', 'TL ', 'TIMOR-LESTE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TGO ', 'TG ', 'TOGO');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TKL ', 'TK ', 'TOKELAU');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TON ', 'TO ', 'TONGA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TTO ', 'TT ', 'TRINITE-ET-TOBAGO ');
```

```
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TUN ', 'TN ', 'TUNISIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TKM ', 'TM ', 'TURKMENISTAN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TCA ', 'TC ', 'TURKS ET CAIQUES, ILES');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TUR ', 'TR ', 'TURQUIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('TUV ', 'TV ', 'TUVALU');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('UKR ', 'UA ', 'UKRAINE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('URY ', 'UY ', 'URUGUAY');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VUT ', 'VU ', 'VANUATU');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VEN ', 'VE ', 'VENEZUELA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('VNM ', 'VN ', 'VIET NAM');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('WLF ', 'WF ', 'WALLIS-ET-FUTUNA');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('YEM ', 'YE ', 'YEMEN ');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ZMB ', 'ZM ', 'ZAMBIE');
INSERT INTO MYLIBRARY.LSTPAYS VALUES ('ZWE ', 'ZW ', 'ZIMBABWE');
```

12.2 Les types de données de DB2

Important : avant de choisir un type SMALLINT ou INTEGER, il est important de noter que le type DECIMAL avec zéro décimales peut constituer une alternative intéressante, car il permet d'obtenir la même précision tout en permettant de définir des valeurs mini et maxi supérieures :

Type	Précision	Valeurs limites mixi maxi
SMALLINT	5	–32.768 à 32.767
DECIMAL	5, 0	–99.999 à 99.999
INTEGER	10	–2.147.483.648 à 2.147.483.647
DECIMAL	10, 0	–9.999.999.999 à 9.999.999.999

Types de données DB2 standards non numériques	
Type de donnée	**Longueur maximum**
CHAR	LUW: 254 octets IBM i: 32.766 octets zSeries: 255 octets
VARCHAR	LUW: 32.672 octets IBM i: 32.740 octets zSeries: 32.704 octets
LONG VARCHAR (LUW seulement)	32.700 octets
CLOB	2.147.483.647 octets
GRAPHIC	LUW: 127 car. IBM i: 16.383 car. zSeries: 127 car.
VARGRAPHIC	LUW: 16.336 car. IBM i: 16.370 car. zSeries: 16.352 car.
DBCLOB	1.073.741.823 car.
BINARY (IBM i seulement)	32.766 octets
VARBINARY (IBM i seulement)	32.740 octets
BLOB	2.147.483.647 octets

Les types de données numériques standards de DB2		
Type de donnée	**Précision (chiffres)**	**Valeurs limites mini et maxi**
SMALLINT (16 bit)	5	–32.768 à 32.767
INTEGER (32 bit)	10	–2.147.483.648 à 2.147.483.647
BIGINT (64 bit)	19	–9.223.372.036.854.775,808 à 9.223.372.036.854.775.807
DECIMAL (packed), NUMERIC	LUW, zSeries: 31 IBM i: 63	LUW, zSeries: toute valeur de 1 à 31 chiffres. IBM i: toute valeur de 1 à 63 chiffres.
REAL	LUW, IBM i: 24 zSeries: 21	LUW: Smallest REAL value –3.402E+38 Largest REAL value +3.402E+38 Smallest positive REAL value +1.175E-37 Largest negative REAL value –1.175E-37 IBM i: Smallest REAL value –3.4E+38 Largest REAL value +3.4E+38 Smallest positive REAL value +1.18E-38 Largest negative REAL value –1.18E-38 zSeries: Smallest REAL value –7.2E+75 Largest REAL value +7.2E+75 Smallest positive REAL value +5.4E–79 Largest negative REAL value –5.4E–79
DOUBLE	53	LUW: Smallest DOUBLE value –1.79769E+308 Largest DOUBLE value +1.79769E+308 Smallest positive DOUBLE value +2.225E–307 Largest negative DOUBLE value –2.225E–307 IBM i: Smallest DOUBLE value –1.79E+308 Largest DOUBLE value +1.79E+308 Smallest positive DOUBLE value +2.23E–308 Largest negative DOUBLE value –2.23E–308 zSeries: Smallest REAL value –7.2E+75 Largest REAL value +7.2E+75 Smallest positive REAL value +5.4E–79 Largest negative REAL value –5.4E–79

Types de données DB2 standards pour les dates, heure et horodatages		
Description	**Valeur mini**	**Valeur maxi**
DATE	0001-01-01	9999-12-31
TIME	00:00:00	24:00:00
TIMESTAMP	0001-01-01-00.00.00.000000	9999-12-31-24.00.00.000000