

Guide de survie SQLfast

MODÈLES TYPIQUES DE SCRIPTS SQLFAST

Décembre 2022

Objectif

Le contenu du présent document est la conversion du fichier d'aide intégré **Help > Survival guide** de SQLfast au niveau **Expert**.

Ce tutoriel propose une série de modèles généraux de scripts SQLfast. Ces modèles peuvent servir de base à des scripts plus spécifiques et plus complexes. Les chapitres du manuel **SQLfast-manual.pdf**, disponible sur le site SQLfast fourniront des exemples et des discussions plus détaillés.

Sauf rares exceptions, les modèles sont illustrés par le traitement de la base de données CLICOM.db, dont le contenu est rappelé ci-après.

Rappel concernant la version d'aide intégrée (**Help > Survival guide**):

- La plupart des modèles sont complets. Ils peuvent être transférés (bouton **Transfer**) directement dans la fenêtre principale pour devenir un script local prêt à être modifié, personnalisé et exécuté.
- Tous les modèles de script sont exprimés sous la forme d'*embedded scripts*. Pour les exécuter, il suffit de cliquer sur leur bouton "**Run**".

Table des matières

- 0. La base de données CLICOM.db - Version Juin 2016**
- 1. Lire les données d'une table**
- 2. Créer une base de données**
- 3. Créer une table temporaire dans une base de données existante**
- 4. Ecrire une ligne de texte**
- 5. Utilisation des variables**
- 6. Valeur fournie par l'utilisateur**
- 7. Saisie et affichage de valeurs multiples**
- 8. Modification de valeurs par l'utilisateur**
- 9. Saisie de valeurs par sélection dans une liste**
- 10. Sélection d'items (unique, multiples)**
- 11. Affichage de valeurs**
- 12. Saisie et affichage d'un texte. Affichage d'un message**
- 13. Le temps**
- 14. Où sommes-nous ? Les variables de l'environnement**
- 15. Les calculs**
- 16. L'alternative**
- 17. Extraction de données d'une ligne d'une table**
- 18. La boucle d'intervalle**
- 19. La boucle SQL**
- 20. La boucle sur fichier de texte**
- 21. La boucle "while"**
- 22. Contrôle d'une boucle**
- 23. Contrôle des dialogues**
- 24. Procédures, fonctions, applications SQLfast. Programmes externes**
 - a) Scripts SQLfast**
 - b) Programmes et fonctions Python**
 - c) Programmes externes**
- 25. Apostrophes, valeurs null et chaînes vides**
- 26. Majuscules, minuscules et accents**
- 27. Affichage du résultat d'une requête**

- 28. Personnalisation du format d'affichage
- 29. Où écrire ? Les différents supports d'écriture
- 30. Les opérations sur les fichiers externes
 - a) Sélection d'un fichier ou d'un répertoire
 - b) Manipulation de fichiers
 - c) Transfert de/vers un fichier
 - d) Lire et écrire dans un fichier de texte
 - e) Consultation d'un répertoire
 - f) Recherche des fichiers homonymes
- 31. Manipulation d'images
- 32. Gestion des erreurs
- 33. Arrêt d'un script
- 34. Contrôle de l'opérateur de substitution
- 35. Contrôle de l'état d'un script
- 36. Exécution automatique d'un script
- 37. Exécution de scripts de grande taille
- 38. Génération-exécution de scripts
- 39. Les tables du dictionnaire de la base de données
- 40. Visualisation des données en HTML
- 41. Ajout d'un mode d'emploi à une application SQLfast
- 42. Extraction de données d'un site web
- 43. Données temporaires : table ou liste SQLfast ?
 - a) Rappel : la liste SQLfast
 - b) Création d'une liste SQLfast
 - c) Manipulation de listes SQLfast (rappel)
 - d) Boucle sur liste
 - e) Exemples d'application
- 44. Tables sans schéma : les colonnes dynamiques
 - a) Les colonnes dynamiques
 - b) Conditions d'utilisation
 - c) Métadonnées des colonnes dynamiques
 - d) Note technique

0. La base de données CLICOM.db - Version Juin 2016

7% Table CLIENT						
primary key [16 rows] Select						
	NCLI	NOM	ADRESSE	LOCALITE	CAT	COMPTE
1	B062	GOFFIN	72, r. de la Gare	Namur	B2	-3200
2	B112	HANSENNE	23, r. Dumont	Poitiers	C1	1250
3	B332	MONTI	112, r. Neuve	Genève	B2	0
4	B512	GILLET	14, r. de l'Eté	Toulouse	B1	-8700
5	C003	AVRON	8, ch. de la Cure	Toulouse	B1	-1700
6	C123	MERCIER	25, r. Lemaître	Namur	C1	-2300
7	C400	FERARD	65, r. du Tertre	Poitiers	B2	350
8	D063	MERCIER	201, bvd du Nord	Toulouse		-2250
9	F010	TOUSSAINT	5, r. Godefroid	Poitiers	C1	0
10	F011	PONCELET	17, Clôs des Erables	Toulouse	B2	0
11	F400	JACOB	78, ch. du Moulin	Bruxelles	C2	0
12	K111	VANBIST	180, r. Florimont	Lille	B1	720
13	K729	NEUMAN	40, r. Bransart	Toulouse		0
14	L422	FRANCK	60, r. de Wépion	Namur	C1	0
15	S127	VANDERKA	3, av. des Roses	Namur	C1	-4580
16	S712	GUILLAUME	14a, ch. des Roses	Paris	B1	0
Insert a row Update a row Delete a row						

7% Table COMMANDE			
primary key [7 rows] Select			
	NCOM	NCLI	DATECOM
1	30178	K111	2015-12-21
2	30179	C400	2015-12-22
3	30182	S127	2015-12-23
4	30184	C400	2015-12-23
5	30185	F011	2016-01-02
6	30186	C400	2016-01-02
7	30188	B512	2016-01-03
Insert a row Update a row Delete a row			

7% Table PRODUIT			
primary key [7 rows] Select			
	NPRO	LIBELLE	PRIX QSTOCK
1	CS262	CHEV. SAPIN 200x6x2	75 45
2	CS264	CHEV. SAPIN 200x6x4	120 2690
3	CS464	CHEV. SAPIN 400x6x4	220 450
4	PA45	POINTE ACIER 45 (1K)	105 580
5	PA60	POINTE ACIER 60 (1K)	95 134
6	PH222	PL. HETRE 200x20x2	230 782
7	PS222	PL. SAPIN 200x20x2	185 1220
Insert a row Update a row Delete a row			

7% Table DETAIL			
primary key [14 rows] Select			
	NCOM	NPRO	QCOM
1	30178	CS464	25
2	30179	CS262	60
3	30179	PA60	20
4	30182	PA60	30
5	30184	CS464	120
6	30184	PA45	20
7	30185	CS464	260
8	30185	PA60	15
9	30185	PS222	600
10	30186	PA45	3
11	30188	CS464	180
12	30188	PA45	22
13	30188	PA60	70
14	30188	PH222	92
Insert a row Update a row Delete a row			

1. Lire les données d'une table

Examiner les données d'une table d'une base de données.

```
openDB CLICOM.db;  
  select * from CLIENT where LOCALITE = 'Poitiers';  
  select * from DETAIL where NPRO = 'PA45';  
closeDB;
```

La dernière instruction (`closeDB`) est recommandée mais est facultative en mode standard, dans lequel la base de données laissée ouverte par le script est automatiquement fermée (paramètre "`autoCloseDB = mainscript`"). On peut donc écrire :

```
openDB CLICOM.db;  
select * from CLIENT;
```

Une instruction peut s'écrire en une ou plusieurs lignes :

```
select NOM,LOCALITE from CLIENT where CAT = 'B1' order by NOM;  
ou  
select NOM,LOCALITE  
from   CLIENT  
where  CAT = 'B1'  
order by NOM;
```

Les lignes de commentaire sont autorisées entre deux fragments d'une instruction mais pas les lignes blanches.

2. Créer et supprimer une base de données

On crée la base de données `VENTES.db`, contenant la table `CLIENT`, dans laquelle on insère quelques lignes.

```
createOrReplaceDB VENTES.db;  
create table CLIENT(NCLI char(10),NOM char(32),  
                    LOCALITE char(30));  
  insert into CLIENT values('B112','HANSENNE','Poitiers');  
  insert into CLIENT values('B512','GILLET','Toulouse');  
  insert into CLIENT values('C400','FERARD','Poitiers');  
  insert into CLIENT values('B332','MONTI','Genève');  
commitDB;  
closeDB;
```

L'instruction "`createOrReplaceDB`" crée une base de données si elle n'existe pas encore et la remplace si elle existe. L'instruction "`createDB`" suppose que la base de données n'existe pas. Dans le cas contraire, une erreur est déclenchée. L'instruction "`commitDB`" rend les modifications persistantes. Sans elle, les modifications seront annulées à la fermeture de la base de données.

Pour supprimer une base de données (ainsi que sa sauvegarde et son fichier de positionnement) :

```
deleteDB VENTES.db;
```

3. Créer une table temporaire dans une base de données existante

On ajoute à la base de données CLICOM.db une table nommée BONS_CLIENT qui disparaîtra automatiquement lors de la fermeture de cette base de données. L'instruction "commitDB" est donc inutile.

```
openDB CLICOM.db;
create temporary table BON_CLIENT(NCLI char(10),
                                   NOM char(32));
insert into BON_CLIENT
select NCLI,NOM from CLIENT where COMPTE > 0;
select * from BON_CLIENT;
closeDB;
```

4. Ecrire une ligne de texte

L'instruction "write" écrit dans la fenêtre de sortie (plus précisément le *canal de sortie*, par défaut la *fenêtre de sortie*) une chaîne de caractères.

```
write Données de la table CLIENT;
write -----;
write Note : ces données sont présentées sous une forme
tabulaire.;
```

Cette chaîne peut être écrite en plusieurs lignes. Cependant, les sauts de lignes sont remplacés par un espace. Ainsi,

```
write Première ligne
Deuxième ligne
Troisième ligne;
```

donne le résultat suivant :

```
Première ligne Deuxième ligne Troisième ligne
```

Pour forcer un saut de ligne, on utilisera la commande '@n' :

```
write Première ligne@nDeuxième ligne@nTroisième ligne;
```

Ce qui donne :

```
Première ligne
Deuxième ligne
Troisième ligne
```

Les espaces de tête de la chaîne sont supprimés. Les deux instructions suivantes produisent le même résultat :

```
write Les clients de Toulouse;
write      Les clients de Toulouse;
```

Pour imposer des espaces en début de chaîne, on utilisera soit la commande de tabulation '@t', soit la commande d'espace '@S5' :

```
write @t@tLes clients de Toulouse;
write @S5Les clients de Toulouse;
```

5. Utilisation des variables

La chaîne 'Poitiers' (sans les apostrophes) est assignée à la variable de nom "ville" (sans les guillemets). Toute occurrence de l'expression \$ville\$ dans les instructions suivantes sera remplacée par la valeur de la variable ville.

```
openDB CLICOM.db;
set ville = Poitiers;
write Clients habitant à $ville$ ;;
select NCLI,NOM from CLIENT where LOCALITE = '$ville$';
```

Ce script produit le résultat suivant :

```
Clients habitant à Poitiers :
+-----+-----+
| NCLI | NOM       |
+-----+-----+
| B112 | HANSENNE  |
| F010 | TOUSSAINT |
| C400 | FERARD    |
+-----+-----+
```

Les délimiteurs \$ et \$ entourant le nom de la variable constituent l'*opérateur de substitution variable/valeur*. L'expression \$ville\$ est remplacée avant exécution par la valeur de ville, soit Poitiers. Ainsi, l'instruction :

```
select NCLI,NOM from CLIENT where LOCALITE = '$ville$'
```

est d'abord transformée par substitution en :

```
select NCLI,NOM from CLIENT where LOCALITE = 'Poitiers'
```

puis est exécutée.

Pour économiser les instructions write, on peut demander que la ligne écrite soit précédée d'une ligne blanche (-b = "before") :

```
write-b Clients habitant à $ville$ ;;
```

ou suivie d'une ligne blanche (-a = "after") :

```
write-a Clients habitant à $ville$ ;;
```

ou les deux (-ab ou -ba):

```
write-ab Clients habitant à $ville$ ;;
```

Les symboles \$ et \$ sont les délimiteurs par défaut. Ceux-ci peuvent être modifiés, notamment par l'instruction `parameter`, qui permet d'écrire :

```
parameter delimiters = [ ];  
write Clients habitant à [ville] ;;
```

En fait, seul le délimiteur de gauche est obligatoire :

```
parameter delimiters = : ;  
write Clients habitant à :ville ;;
```

Ou, pour les plus littéraires :

```
parameter delimiters = valeur( ) ;  
write Clients habitant à valeur(ville) ;;
```

Une instruction SQLfast peut être partiellement ou complètement définie à partir de variables. Quelques exemples :

```
set select = NCLI,NOM;  
set table = CLIENT;  
openDB CLICOM.db;  
select $select$ from $table$;  
closeDB;  
  
set open = openDB CLICOM.db;  
set sfw = select * from CLIENT;  
set close = closeDB;  
$open$;  
$sfw$;  
$close$;
```

Si la variable `var` n'existe pas encore lors de l'exécution d'une instruction qui inclut l'expression `$var$`, cette dernière restera inchangée.

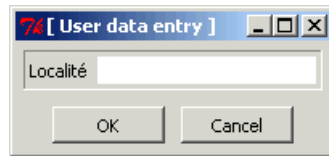
6. Valeur fournie par l'utilisateur

Il est possible de demander à l'utilisateur d'un script de fournir la valeur d'une variable par l'instruction `ask`:

```
ask ville = Localité;  
write Clients habitant à $ville$ ;;
```

Une boîte de saisie de données s'ouvre et invite l'utilisateur à introduire le nom d'une localité. L'étiquette `Localité` est assignée au champ de saisie. La valeur obtenue est assignée à la variable de réception `ville`.

La variable `DIALOGbutton` indique le bouton utilisé pour quitter la boîte ('OK' ou 'Cancel'). La variable de réception n'est pas modifiée si la boîte est fermée via le bouton 'Cancel'. Ces règles sont applicables à toutes les boîtes de saisie.



Pour les utilisateurs pressés, il existe une forme simplifiée dans laquelle le nom de la variable sert d'étiquette du champ de saisie :

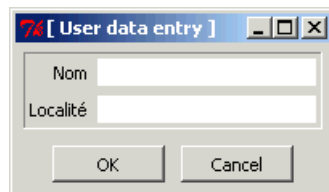
```
ask Localité;
write Clients habitant à $Localité$ ;;
```

7. Saisie de valeurs multiples

Une boîte de saisie à champs multiples permet de collecter plusieurs valeurs en une seule opération :

```
ask nom,ville = Nom|Localité;
write Clients de nom $nom$ habitant à $ville$ ;;
```

Ce qui donne :



Le symbole '|' sépare les étiquettes des différents champs et indique que ceux-ci sont disposés verticalement. Le séparateur '|' les disposeraient horizontalement, côte à côte.

Ici encore, on pourra utiliser la forme simplifiée :

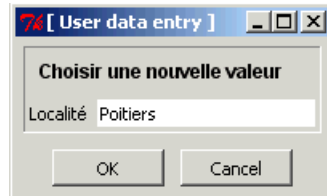
```
ask Nom,Localité;
write Clients de nom $Nom$ habitant à $Localité$ ;;
```

8. Modification et affichage de valeurs par l'utilisateur

L'instruction ask-u (-u pour *update*) propose à l'utilisateur de modifier une valeur :

```
set ville = Poitiers;
ask-u ville = [/bChoisir une nouvelle valeur] Localité;
write Clients de nom $nom$ habitant à $ville$ ;;
```

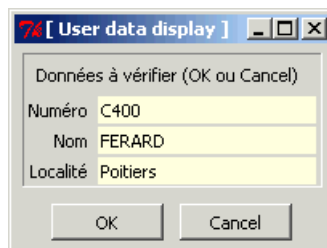
Ce qui donne :



Le bloc [/bChoisir une nouvelle valeur] insère un commentaire destiné, par exemple, à informer l'utilisateur. Le paramètre /b indique que ce commentaire doit apparaître en gras (b pour *boldface*). Le commentaire peut occuper un nombre quelconque de lignes. Un passage à la ligne s'indique par l'insertion de la commande '@n' dans le commentaire. De même, la commande '@t' introduit une tabulation.

L'instruction `showData` crée une boîte semblable à celle de `ask-u`, mais ne permet pas la modification des valeurs des champs. Elle est utilisée pour afficher une collection de valeurs :

```
set cli = C400;
set nom = FERARD;
set loc = Poitiers;
showData cli,nom,loc = [Données à vérifier (OK ou Cancel)]
                      Numéro|Nom|Localité;
```



Les formes simplifiées s'appliquent également à ces instructions :

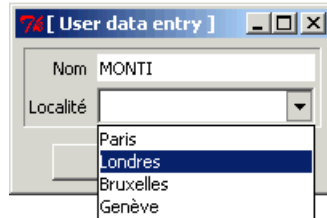
```
ask-u Nom,Localité;
showData Numéro,Nom,Localité;
```

9. Saisie de valeurs par sélection dans une liste

Il s'agit d'inviter l'utilisateur à sélectionner une valeur dans une liste de propositions prédéfinies au lieu d'introduire une valeur au clavier :

```
ask nom,ville =
    Nom|Localité[(Paris,Londres,Bruxelles,Genève)];
write Clients de nom $nom$ habitant à $ville$ ;;
```

Le bloc [(Paris,Londres,Bruxelles,Genève)] spécifie la liste (entre parenthèses) de valeurs de référence. A l'exécution, une liste déroulante apparaît:



Les valeurs peuvent être extraites de la base de données :

```
openDB CLICOM.db;
ask ville = Localité[select distinct LOCALITE
                    from CLIENT order by LOCALITE];
write Clients habitant à $ville$ ;
```

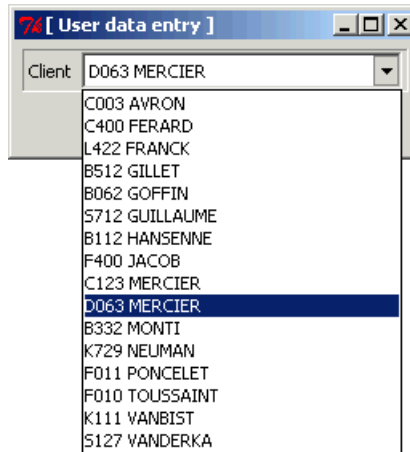
Le bloc [select distinct LOCALITE from CLIENT] spécifie la liste de valeurs sous la forme de requête SQL. Si la liste est longue, il est prudent de la trier par une clause "order by".

Dans les deux formats de liste, l'utilisateur peut également introduire une valeur au clavier. On peut cependant interdire la saisie manuelle en forçant l'utilisateur à sélectionner une valeur dans la liste. On utilisera pour ce faire le symbole '!':

```
openDB CLICOM.db;
ask nom,ville = Nom
                |Localité[!(Paris,Londres,Bruxelles,Genève)];
ask ville = Localité[select distinct LOCALITE
                    from CLIENT order by LOCALITE];
```

Il est possible de distinguer les valeurs apparaissant dans la liste de celles qui sont assignées à la variable. Dans l'exemple suivant, la liste reprend le numéro et le nom des clients alors que la variable ne reçoit que le numéro. Ceci facilite à la fois la sélection d'une valeur par l'utilisateur et le traitement de la valeur reçue par la variable :

```
openDB CLICOM.db;
ask ncli = Client[select NCLI||'|'||NOM, NCLI
                  from CLIENT order by NOM,NCLI];
select * from CLIENT where NCLI = '$ncli$';
```



10. Sélection d'items (unique, multiple)

L'instruction `selectOne` crée une boîte comportant un message (optionnel) et une liste de boutons radio. La variable de réception (`nTable`) contient le numéro de l'item coché.

```
selectOne nTable = [/bChoisir une table à visualiser]
                  CLIENT|PRODUIT|COMMANDE|DETAIL;
write La table $nTable$ a été sélectionnée.;
```



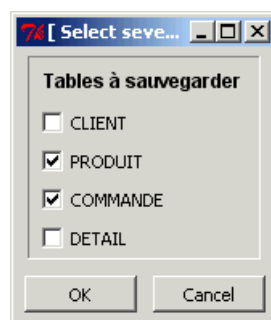
Si nous désirons le nom associé à cette case, nous procéderons comme suit, en utilisant la fonction `item`. Cette fonction extrait d'une liste (ici la valeur de `lTable`) un élément d'indice donné (ici la valeur de `nTable`). La liste est une chaîne de caractères dont les items sont séparée par un symbole particulier (ici '|') :

```
set lTable = CLIENT|PRODUIT|COMMANDE|DETAIL;
selectOne nTable = [Choisir une table à visualiser] $lTable$;
compute nomTable = item('$lTable$', $nTable$, '|');
```

```
write La table $nomTable$ a été sélectionnée.;
```

L'instruction `selectMany` crée une boîte comportant un message (optionnel) et une liste de cases à cocher. A chaque case correspond une variable de réception (`cli`, `pro`, `com`, `det`) indiquant (0/1) si l'item a été coché.

```
selectMany cli,pro,com,det = [Tables à sauvegarder]
                                CLIENT|PRODUIT|COMMANDE|DETAIL;
write Tables sélectionnées : $cli$, $pro$, $com$, $det$;
```



La conversion des numéros de cases cochées en noms de table pourrait se coder comme suit :

```
set lTable = CLIENT|PRODUIT|COMMANDE|DETAIL;
selectMany t1,t2,t3,t4 = [Tables à sauvegarder] $lTable$;
for nTable = [1,4];
    if ($t$nTable$$ = 0) next;
    compute nomTable = item('$lTable$', $nTable$, '|');
    write La table $nomTable$ a été sélectionnée;
endfor;
```

Les noms des variables correspondant aux cases à cocher (`cli`, `pro`, `com`, `det`) sont remplacés par la lettre 't' suivie d'un indice de 1 à 4 (`t1`, `t2`, `t3`, `t4`). La boucle teste la valeur de chaque variable et écrit un message si cette valeur n'est pas 0. L'expression `$t$numTable$$` demande deux substitutions : `$t$numTable$$` est d'abord convertie en `$t3$` (si `$numTable$ = 3`), ensuite convertie en 0 ou 1.

Ces instructions existent en variante `update` :

```
selectOne-u et selectMany-u.
```

11. Saisie et affichage d'un texte. Affichage d'un message

La saisie d'un texte de longueur quelconque s'effectue par l'instruction `askText`. Celle-ci crée une boîte de saisie éventuellement accompagnée d'un message.

Si un message n'est pas nécessaire, on écrira :

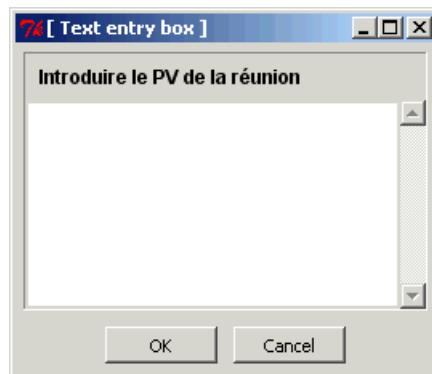
```
askText pv;
write $pv$;
```

Et, avec un message :

```
askText pv = [Introduire le PV de la réunion];
write $pv$;
```

Le bloc de message peut spécifier la taille de la fenêtre lorsque la taille par défaut (20 lignes de 64 caractères) ne convient pas : **/x32** pour une largeur de 32 caractères et **/y8** pour une hauteur de 8 lignes

```
askText pv = [/b/x32/y8Introduire le PV de la réunion];
write $pv$;
```



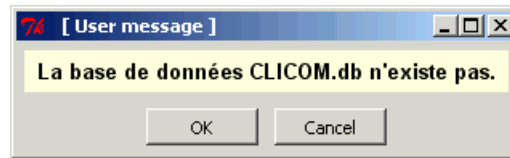
Cette instruction existe aussi en version update : `askText-u` :

```
set pv = PV de la réunion du 2 mars\nPoint 1:\nPoint 2;;
askText-u pv;
write $pv$;
```

On notera l'usage du symbole `'\n'` pour forcer le retour à la ligne. Le texte n'étant pas mis en forme par SQLfast mais simplement inséré dans zone de texte, le symbole `'@n'` n'est pas interprété.

Pour un affichage en lecture seule, on utilisera l'instruction `showText`. L'affichage d'un message court se fera plus simplement par l'instruction `showMessage` aussi nommée `pause` :

```
set db = Com.db;
showMessage La base de données $db$ n'existe pas.;
pause Je répète : la base de données $db$ n'existe pas.;
```



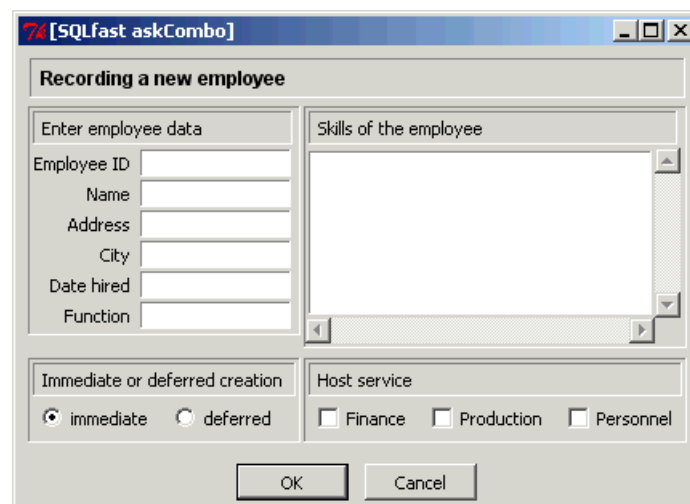
12. Les boîtes de saisie/affichage composites

Une boîte composite est constituée d'un assemblage de boîtes de dialogues élémentaires : ask, ask-u, showData, askText, askText-u, showText, selectOne, selectOne-u, selectMany, selectMany-u et showPicture.

Elle est créée par l'instruction askCombo, qui spécifie un message (optionnel) ainsi qu'une liste de définitions de boîtes élémentaires chacune entre crochets. Les séparateurs | et || définissent l'agencement en lignes et en colonnes des boîtes élémentaires.

Pour la construction des boîtes complexes, il est conseillé de ranger la définition de chaque boîte élémentaire dans une variable puis de définir la boîte composite à partir des valeurs de ces variables :

```
set D = ask eid,nam,add,cit,dat,fun = [Enter employee data]
      Employee ID|Name|Address|City|Date hired|Function;
set O = selectOne delay = [Immediate or deferred creation?]
      immediate || deferred;
set T = askText ski = [/w0/y7/x26 Skills of the employee];
set M = selectMany fin,pro,per = [Host service]
      Finance|Production|Personnel;
askCombo [/b Recording an employee] [$D$] | [$O$] || [$T$] | [$M$];
```



Cette boîte a été obtenue pour une longueur des champs de saisie de 15 caractères :

```
parameter entryWidth = 15;
```

13. Le temps

SQLfast (conforme en cela à SQL) reconnaît les types temporels suivants :

- *date* : chaîne de caractères au format '2016-08-24'
- *time* : chaîne de caractères au format '22:45:06' ou, incluant les millisecondes : '22:45:06.083'
- *datetime* : combinaison de date et temps : '2016-08-24 22:45:06'
- *period*: nombre de secondes et millisecondes : 142.109

Les *date*, *temps* et *durée* courants sont disponibles via trois variables systèmes : *date/date1* (date courante), *time* (temps dans la journée) et *timer* (temps écoulé, en secondes et millisecondes, depuis remise à zéro du chrono).

L'instruction "start-timer" remet le chrono (*timer*) à zéro. L'instruction *wait* suspend l'exécution soit un nombre fixe de millisecondes (*wait 500*), soit un temps aléatoire entre deux bornes (*wait 100,900*).

```
write Date   : $date$;
write Date1  : $date1$;
write Temps  : $time$;
start-timer;
wait 100,900;
write Durée  : $timer$;
```

Résultat :

```
Date   : 2016-2-18
Date1  : 18-02-2016
Temps  : 22:58:54.703
Durée  : 0.453
```

Ces variables sont utiles pour dater un résultat et pour mesurer le temps d'exécution d'un traitement :

```
openDB CLICOM.db;
start-timer;
select count(*) from CLIENT,COMMANDE,DETAIL,PRODUIT;
write Temps d'exécution : $timer$;
closeDB;
```

Les constantes temporelles SQL sont aussi disponibles dans l'instruction *compute*, dans les conditions *et*, bien sûr dans les requêtes SQL :

```
current_date, current_time et current_timestamp
```

S'y ajoute la fonction SQL *current_timestamp_full()* qui complète la constante *current_timestamp* en y ajoutant es millisecondes du temps courant. Les fonctions *year*, *month*, *day_in_month*, *hour*, *minute*, *second*, *millisecond*

extraient les composants de ces constantes, variables et fonctions (sauf `timer`, dont la valeur d'obéit pas au format standard).

Exemples d'utilisation :

```
compute T = current_timestamp_full();
compute Y = year('$T$');
compute M = month('$T$');
compute D = day_in_month('$T$');
compute h = hour('$T$');
compute m = minute('$T$');
compute s = second('$T$');
compute ms = millisecond('$T$');
write Le temps "$T$" se décompose en ($Y$, $M$, $D$)
      et ($h$, $m$, $s$, $ms$);
```

Résultat :

```
Le temps 2016-10-25 23:33:13.953 se décompose en 2016,10,25
et 23,33,13,953
```

Calculs temporels

Les calculs sur des valeurs temporelles *date*, *time* et *datetime* sont très complexes et dépendent du calendrier utilisé. Pour faciliter et uniformiser ces calculs, on convertit ces valeurs en *jours juliens*. Ce système représente toute valeur de date ou datetime par un nombre réel dont 0.0 désigne le 24 novembre -4712 à midi. La partie entière du jour julien représente un jour et la partie décimale un temps dans la journée. Cependant, la représentation et les calculs ne sont garantis corrects que dans l'intervalle [0000-01-01 00:00:00 , 9999-12-31 23:59:59].

Les conversions s'effectuent via la fonction `julianday` dans un sens et les fonctions `date`, `time` et `datetime` dans le sens inverse :

```
compute J = julianday('2016-07-24 16:04:55');
compute D = date($J$);
compute T = time($J$);
compute DT = datetime($J$);
write $J$, [$D$], [$T$], [$DT$];
```

Ce qui donne :

```
2457594.17008, [2016-07-24], [16:04:54], [2016-07-24 16:04:54]
```

La différence entre deux dates s'exprime donc comme suit :

```
compute DD = int(julianday('2016-07-24')
- julianday('2016-02-24'));
write Entre '2016-02-24' et '2016-07-24' il y a $DD$ jours;
```

Ce qui donne :

```
Entre '2016-02-24' et '2016-07-24' il y a 151 jours
```

D'ailleurs, on retrouve la date '2016-07-24' comme suit :

```
compute D2 = date(julianday('2016-02-24') + 151);
```

Le calcul du temps s'effectue de manière semblable :

```
compute IT = julianday('23:59:59') - julianday('11:59:59');
write Intervalle = $IT$;
```

Soit :

```
Intervalle = 0.5;
```

Comment ajouter un temps, par exemple 8h30, à une date ? On serait tenté d'écrire :

```
compute D4 = datetime(julianday('2016-10-32 18:00:00') +
julianday('08:30:00'));
write Résultat = $D4$;
```

Manifestement cela ne marche pas.

Il faut savoir que lorsque la date est absente d'une valeur de datetime, la date conventionnelle 2000-01-01 est ajoutée. Le temps 08:30:00 est donc en réalité la date 2000-01-01 08:30:00. Pour exprimer 08:30:00 seul, il faut retirer de son jour julien celui de 2000-01-01. On devra donc écrire :

```
compute t = julianday('08:30:00') - julianday('2000-01-01');
compute D4 = datetime(julianday('2016-10-31 18:00:00') + $t$);
write Résultat = $D4$;
```

14. Où sommes-nous ? Les variables de l'environnement

SQLfast dispose d'une série de variables systèmes renseignant sur le contexte d'exécution d'un script : les répertoires courants, la plateforme, le SGBD, la base de données ouverte, le nom du script en cours d'exécution, le niveau d'appel du script courant, etc. :

```
openDB CLICOM.db;
write scriptDirectory : $scriptDirectory$;
write dbDirectory : $dbDirectory$;
write platform : $platform$;
write DBMS : $DBMS$;
write dbName : $dbName$;
write scriptName : $scriptName$;
closeDB;
```

Ce qui nous donnera un aperçu de l'environnement courant:

```
scriptDirectory : D:/SQLfast-2015/SQLfast/Chapitre-18
dbDirectory : D:/SQLfast-2015/Databases/
platform : win32
DBMS : SQLite 3
dbName : CLICOM.db
```

```
scriptName :      _SQLfast_local.sql
```

Remarque

`_SQLfast_local.sql` est le nom du fichier dans lequel est stocké le script soumis à exécution via l'interface graphique. On évitera d'attribuer ce nom à un *vrai* script.

15. Les calculs

L'instruction "compute" assigne à une variable le résultat de l'évaluation d'une expression SQL.

```
set N = 127;
compute N = $N$ + 1;
compute dist = sqrt($N*$N$ + 1) - sqrt($N*$N$ - 1);
write N = $N$; dist = $dist$;

set nom = DUPONT;
set ville = Genève;
compute adresse = upper(substr('$nom$',1,1))
                  |lower(substr('$nom$',2,99))
                  |' '
                  |simplify('$ville$');
write-b $adresse$;
```

Ce qui donne :

```
N = 128; dist = 0.00781250000364
Dupont geneve
```

Remarque importante

L'expression à évaluer obéit strictement à la syntaxe SQL. Néanmoins, contrairement aux requêtes SQL proprement dites, les fonctions admises sont indépendantes du SGBD utilisé. Les assignations suivantes sont valides :

```
compute OK = $adresse$ like '%Neuve%'
compute année = year(current_date)
compute bonus = case when $M$ < 1000 then 20 else $M$/50 end;
compute traduc = replace(trim($adresse$), 'Neuve', 'New');
```

Les fonctions admises en SQLfast sont répertoriées dans le tutoriel *SQLfast commands* (en anglais) ainsi que dans le document *SQLfast-Commands.pdf* qui en dérive.

16. L'alternative

L'alternative, ou instruction `if`, permet d'exécuter une ou plusieurs instructions sous condition.

Dans cet exemple, la suppression de toutes les lignes de la table DETAIL sera effectuée si l'utilisateur le demande explicitement (réponse 1) :

```
openDB CLICOM.db;
ask del = Supprimer lignes de DETAIL (0/1) ?;
if ($del$ = 1) delete from DETAIL;
select * from DETAIL;
abortDB;
closeDB;
```

On a annulé la modification (abortDB) de manière à préserver l'état initial de la base de données.

L'exécution conditionnelle peut concerner une séquence de plusieurs instructions :

```
openDB CLICOM.db;
ask del = Supprimer lignes de DETAIL (0/1) ?;
if ($del$ = 1);
    delete from DETAIL;
    showMessage La table DETAIL est vide;
endif;
abortDB;
closeDB;
```

On peut aussi spécifier les opérations à exécuter si la condition est fausse :

```
openDB CLICOM.db;
ask del = Supprimer lignes de DETAIL (0/1) ?;
if ($del$ = 1);
    delete from DETAIL;
    showMessage La table DETAIL est vide;
else;
    showMessage La table DETAIL n'a pas été modifiée;
endif;
abortDB;
closeDB;
```

Remarque importante

La condition obéit strictement à la syntaxe SQL. On écrira donc :

```
if ('$ville$' = 'Poitiers')
if (upper('$libelle$') like '%SAPIN%')
if ('$nombre$' = '0')
if ($nombre$ = 0)
if (simplify('$localite$') = 'geneve')
if (cast('$nombre$' as integer) = 0)
```

17. Extraction de données d'une ligne d'une table

L'instruction `extract` exécute une requête SQL `select` et range dans les variables de réception les données fournies de la première ligne du résultat. Si ce résultat est vide, les variables de réception seront vides également.

```

openDB CLICOM.db;
extract nom,loc = select NOM,LOCALITE from CLIENT
                  where NCLI = 'C400';
write-b $nom$, $loc$;
extract nom,loc = select NOM,LOCALITE from CLIENT
                  where CAT is not null;
write $nom$, $loc$;
extract nom,loc = select NOM,LOCALITE from CLIENT
                  where CAT is not null #2;
write-a $nom$, $loc$;
closeDB;

```

Remarque importante

Le comportement de `ask` est très différent de celui de `extract` en cas d'absence de valeurs : pas de modification pour `ask` et mise à vide pour `extract`. Ce dernier comportement est justifié par les subtilités liées aux valeurs null et aux ensemble vides.

Cette question est traitée de manière détaillée dans le *Tutoriel SQLfast*, chapitre 8, paragraphe 8.8 *Empty sets, null values and statistics*.

18. La boucle d'intervalle

L'entête de boucle `for` permet d'exécuter une ou plusieurs instructions pour les valeurs successives d'un intervalle de nombres entiers ou réels.

```

for I = [1,5] write I = $I$;
for J = [1,5,2] write J = $J$;
for K = [1,2,0.1] write K = $K$;
for M = [1,5] for N = [1,$M$] write (M,N) = ($M$, $N$);

```

La forme `for-endfor` permet un corps de boucle de plusieurs instructions.

```

for I = [1,5];
  compute résultat = cast($I*$I-1)/2 as real);
  write Résultat pour I = $résultat$;
endfor;

```

19. La boucle SQL

Cette boucle `for` permet d'exécuter une ou plusieurs instructions pour les lignes successives du résultat d'une requête d'extraction SQL.

```

openDB CLICOM.db;
for nom,loc = [select NOM,LOCALITE from CLIENT]
  write $nom$, $loc$;
closeDB;

```

La forme `for-endfor` permet un corps de boucle de plusieurs instructions.

```

openDB CLICOM.db;
ask cat = Catégorie[!select distinct CAT from CLIENT
                      where CAT is not null];
set I = 0;
for nom,loc = [select NOM,LOCALITE from CLIENT
              where CAT = '$cat$'];
  compute I = $I$ + 1;
  write Client $I$: $nom$, $loc$;
endfor;
closeDB;

```

Il est possible de limiter le résultat aux lignes dont le rang tombe dans un intervalle :

```

openDB CLICOM.db;
set de = 2;
set à = 5;
for nom,loc = [select NOM,LOCALITE
              from CLIENT order by NCLI #[1,5]]
  write $nom$, $loc$;

write;
for nom,loc = [select NOM,LOCALITE
              from CLIENT order by NCLI #[$de$, $à$]]
  write $nom$, $loc$;

write;
for nom,loc = [select NOM,LOCALITE
              from CLIENT order by NCLI #[1,5,2]]
  write $nom$, $loc$;

```

Une structure de boucles emboîtées permet la production de rapports personnalisés (la commande "@S3" crée 3 espaces) :

```

openDB CLICOM.db;
for cli,nom = [select NCLI,NOM from CLIENT
              where LOCALITE = 'Poitiers'
              and NCLI in (select NCLI from COMMANDE)];
  write CLIENT : $cli$ $nom$;
  for com,dat = [select NCOM,DATECOM
                  from COMMANDE where NCLI = '$cli$'];
    write @S3COMMANDE $com$ du $dat$ :;
    for pro,qte,pri,mon = [select D.NPRO,QCOM,PRIX,QCOM*PRIX
                              from DETAIL D,PRODUIT P
                              where NCOM = '$com$'
                              and D.NPRO = P.NPRO];
      write @S6Détail ($pro$) : $qte$ x $pri$ = $mon$;
    endfor;
  endfor;
closeDB;

```

Résultat :

```

CLIENT : C400 FERARD
COMMANDE 30179 du 2015-12-22 :
  Détail (CS262) : 60 x 75 = 4500
  Détail (PA60) : 20 x 95 = 1900
COMMANDE 30184 du 2015-12-23 :
  Détail (CS464) : 120 x 220 = 26400
  Détail (PA45) : 20 x 105 = 2100

```

```

COMMANDE 30186 du 2016-01-02 :
    Détail (PA45) : 3 x 105 = 315

```

Exemple de lecture par pages de 5 lignes:

```

openDB CLICOM.db;
extract Ncli = select count(*) from CLIENT;
for de = [1,$Ncli$,5];
    compute à = $de$ + 4;
    for nom,loc = [select NOM,LOCALITE
                  from CLIENT order by NOM #[$de$, $à$]]
        write $nom$, $loc$;
    pause;
endfor;
closeDB;

```

20. La boucle sur fichier de texte

La boucle `for` sur fichier permet d'extraire les lignes successives d'un fichier de texte (on vérifie que le fichier de nom `donnees.txt` se trouve dans le répertoire `Files`).

```
for ligne = [file Files/donnees.txt] write $ligne$;
```

ou

```

for ligne = [file Files/donnees.txt];
    compute item1 = item('$ligne$',1,',');
    compute item2 = item('$ligne$',2,',');
    write $item1$ + $item2$;
endfor;

```

La boucle de lecture d'un fichier peut aussi s'exprimer à l'aide des primitives élémentaires `openTextFile`, `nextTextLine` et `closeTextFile` :

```

openTextFile Files/donnees.txt;
while (True);
    nextTextLine ligne = Files/donnees.txt;
    if ('$FILEdiag$' = 'NONE') exit;
    write $ligne$;
endwhile;
closeTextFile Files/donnees.txt;

```

21. La boucle "while"

La boucle `while` permet d'exécuter une séquence d'une ou plusieurs instructions tant qu'une condition est satisfaite.

```

set I = 0;
while (not $I$ between 1 and 10) ask I = Nombre;;
write $I$;

```

```

openDB CLICOM.db;
while ($I$ <> 0);
    extract A = select LOCALITE
                from CLIENT where NCLI = '$ID$';
    execSQL Process.sql;
endwhile;

```

22. Contrôle d'une boucle

On peut modifier le déroulement naturel d'une boucle par deux instructions :

- next : pour forcer l'itération suivante de la boucle,
- exit : pour sortir prématurément de la boucle.

Une technique (un peu barbare) pour ignorer les clients en négatif :

```

openDB CLICOM.db;
for nom,loc,cpt = [select NOM,LOCALITE,COMPTE from CLIENT];
    if ($cpt$ < 0) next;
    write $nom$, $loc$, $cpt$;
endfor;
closeDB;

```

Une boucle while peut être complètement contrôlée par les instructions next et exit :

```

while (True);
    execSQL Process.sql;
    if ($I$ = 0) exit;
    if ($I$ > 10) next;
    execSQL Print.sql;
endwhile;

```

23. Contrôle des dialogues

A la fermeture d'une boîte de dialogue, la variable système `DIALOGbutton` contient le nom du bouton par lequel l'utilisateur a quitté cette boîte.

```

ask com,pro,qte = Commande:|Produit:|Quantité;;
if ('$DIALOGbutton$' = 'OK')
    insert into DETAIL values ('$com$', '$pro$', $qte$);

```

Le contrôle peut aussi se faire sur les valeurs saisies :

```

ask com,pro,qte = Commande:|Produit:|Quantité;;
if ('$DIALOGbutton$' = 'Cancel' or '$com$' = '') exit;

```

24. Procédures, fonctions, applications SQLfast, programmes externes

SQLfast offre plusieurs méthodes d'appel d'une fonction, d'une procédure ou d'un programme externe à partir d'un script.

Script SQLfast

La première méthode est la demande d'exécution (ou *appel*) d'un script SQLfast via l'instruction `execSQL` :

```
set nomFichier = SQLfast1.ini;
execSQL lireParametres.sql;
write $resultat$;
```

Toutes les variables étant globales, le script appelé a accès à toutes les variables existantes, tant en consultation (`nomFichier`) qu'en modification (`resultat`). Il n'y a pas donc pas de passage explicite d'arguments.

Programme ou fonction Python

La deuxième technique consiste à invoquer, via l'instruction `execApp`, un programme Python en mentionnant son nom (sans extension), la valeur de ses arguments (`$profil$`) et les variables recevant ses résultats (`v1, v2`) :

```
execApp v1,v2 = saisieParametres $profil$;
```

ou encore une fonction de ce programme :

```
execApp v1,v2 = saisieParametres:Standard $profil$;
```

L'instruction `function` est synonyme de `execApp` mais est plus naturelle lorsqu'on n'invoque qu'une fonction de ce programme.

```
function long,statut = LGeo:calculPerimetre $poly$;
```

Remarques

- La première forme de `execApp` mentionnant seulement le nom du programme appelle en réalité la fonction **Main**(arg) de ce programme. Elle est donc équivalente à ceci :

```
execApp v1,v2 = saisieParametres:Main $profil$;
```

- Les arguments et les résultats de l'appel sont des chaînes de caractères.
- Les arguments de l'appel d'une fonction forment une liste de constantes séparées par des virgules. Si une constante inclut elle-même une virgule, l'analyse de la liste risque d'être erronée. Dans ce cas, on entourera la constante à risque d'accolades :

```
execApp v1,v2 = saisieParametres:Main {14, rue de l'Eté};
```

Programme externe

La troisième forme consiste à lancer un programme externe, tel qu'un lecteur pdf, un traitement de texte ou un navigateur :

```
execProg "D:/Program Files/.../Foxit Reader.exe" Manuel.pdf;
execProg "D:/Program Files/.../firefox.exe" Client.html;
```

Le fichier SQLfast.ini permet d'initialiser une suite de variables dédiées aux principaux types de programmes externes :

```
TEXTprocessor = notepad.exe
WSprocessor = "D:/Program Files/.../Office/excel.exe"
DOCprocessor = "D:/Program Files/.../Office/winword.exe"
IMAGEprocessor = "D:/Program Files/.../PSP.exe"
PDFprocessor = "D:/Program Files/.../Foxit Reader.exe"
HTMLprocessor = "D:/Program Files/.../firefox.exe"
VIDEOprocessor = "D:/Program Files/VideoLAN/VLC/vlc.exe"
SOUNDprocessor = "D:/Program Files/VideoLAN/VLC/vlc.exe"
```

On écrira donc plutôt :

```
execProg $HTMLprocessor$ Client.html;
```

Remarque importante

Dès qu'un programme externe est lancé par l'instruction `execProg`, le script appelant continue son exécution sans attendre la terminaison du programme.

25. Apostrophes, valeurs *null* et chaînes vides

Les valeurs issues des boîtes de saisie, les données extraites de la base de données, les valeurs des variables d'un script et les valeurs à stocker dans la base de données obéissent à des règles différentes, quel que soit le SGBD et quel que soit le langage de programmation dans lequel le script est rédigé.

a) Constantes SQL et apostrophes

Prenons comme exemple la valeur de la colonne ADRESSE du client B512, soit

```
14, r. de l'Eté
```

constituée de 15 caractères, y compris une apostrophe interne et une virgule.

C'est cette chaîne qui est enregistrée telle quelle sur le disque, qui est introduite par un utilisateur dans un champ d'une boîte de saisie (`ask`) et qui est affichée à l'écran dans une boîte d'affichage (`showData`) ou de modification (`ask-u`).

Cependant, elle ne peut pas apparaître comme constante dans une instruction qui l'enregistre dans la base de données, car elle ne respecte pas la syntaxe SQL des constantes.

D'une part, elle doit être **entourée d'apostrophes** et d'autre part, les **apostrophes internes doivent être dédoublées**. Cette valeur d'adresse doit être transformée comme suit avant toute utilisation en SQL :

```
'14, r. de l''Eté'
```

On devra donc écrire, dans les requêtes SQL :

```
... where ADRESSE = '14, r. de l''Eté'
```

ou encore :

```
... values ('B512','GILLET','14, r. de l''Eté', ...)
```

Le langage SQLfast propose deux techniques pour conditionner la valeur d'une variable selon la syntaxe SQL.

- Outre l'opérateur de substitution habituel `$var$`, il existe une variante, `$vars$`, qui a pour effet de remplacer le nom de la variable par sa valeur, mais après avoir doublé les apostrophes que cette valeur contient.

On peut donc écrire :

```
set adr = 14, r. de l'Eté;
select * from CLIENT where ADRESSE = '$adr$';
```

- La bibliothèque **LStr** (*String Library*) propose deux fonctions de transformation appropriées, `SQLquote` et `SQLquote2`. `SQLquote` ajoute les apostrophes externes et double les apostrophes internes tandis que `SQLquote2` se contente de doubler les apostrophes internes.

On peut donc écrire :

```
set adr = 14, r. de l'Eté;
function adr = LStr:SQLquote {$adr$};
select * from CLIENT where ADRESSE = $adr$;
```

ou encore:

```
set adr = 14, r. de l'Eté;
function adr = LStr:SQLquote2 {$adr$};
select * from CLIENT where ADRESSE = '$adr$';
```

Les accolades entourant `$adr$` dans l'argument de la fonction `SQLquote` protègent sa valeur et empêchent l'interprétation des virgules éventuelles comme séparateurs d'arguments.

b) Chaînes vides et valeurs *null*

SQL distingue, parmi des chaînes de caractères,

- la **chaîne vide**, dont la longueur est 0, et qui peut être concaténée à d'autres chaînes sans les détruire

- la **chaîne absente**, représentée par une valeur null, dont la longueur est null et dont la concaténation avec d'autres chaîne produit null.

En revanche, les boîtes de saisie et les variables ne connaissent que la chaîne vide.

Deux problèmes : (1) comment introduire des valeurs *null* dans la base de données et (2) comment afficher une valeur null extraite de la base de données.

- Valeurs *null* à introduire dans la base de données

Une valeur renvoyée par une boîte de saisie ne distingue pas les deux cas : un champ laissé vide retourne une chaîne vide, quelle que soit sa signification. Une interprétation raisonnable serait celle-ci :

- si la chaîne vide est à assigner à une colonne obligatoire (not null), elle est interprétée comme une chaîne existante mais vide,
- si la chaîne vide est à assigner à une colonne facultative (nullable), elle est interprétée comme une valeur null.

Le script ci-dessous modifie la valeur de CAT (colonne facultative) du client C400. Il reçoit de la boîte de saisie une chaîne qui peut être vide, auquel cas, elle est interprétée comme null :

```
openDB CLICOM.db;
ask cat = [/bCatégorie du client C400] Catégorie;
update CLIENT
set CAT = case when '$cat$' = '' then null else '$cat$' end
where NCLI = 'C400';
select NCLI,CAT from CLIENT;
abortDB;
closeDB;
```

Pour une colonne numérique facultative, on écrira plutôt (en supposant que la colonne COMPTE soit facultative) :

```
set COMPTE = case when '$cpt$' = '' then null else $cpt$ end
```

- Valeurs *null* extraites de la base de données

Comment afficher une valeur *null* ? Comment l'assigner à une variable ?

Si le résultat d'un `select` contient des valeurs null, celles-ci sont transformées selon le paramètre `dispNull` du fichier d'initialisation `SQLfast.ini` :

```
dispNull = --
```

La valeur par défaut est une chaîne vide. L'assignation ci-dessus la remplace par `--`, jugé plus *esthétique* à l'écran !

Si une valeur *null* est assignée à une variable dans une instruction `extract` ou `for-SQL`, elle est remplacée par une **chaîne vide**.

26. Majuscules, minuscules et accents

La présentation triée de chaînes de caractères rencontre quelques problèmes lorsque celles-ci comportent des majuscules, des minuscules et les lettres accentuées. En effet, des lettres telles que 'a', 'A', 'À' et 'à' n'apparaissent pas consécutives. Selon les schémas de codage courants (qui sont pour la plupart des extensions du code ASCII 7 bits), le bloc des minuscules suit celui des majuscules et précède celui des lettres accentuées, de sorte que 'Z' < 'a', et 'z' < 'à'.

Par conséquent, la requête SQL :

```
select * from CLIENT order by NOM;
```

lorsque les noms de clients comprennent des lettres minuscules, majuscules et accentuées, produit généralement un résultat peu lisible.

Une solution consiste à trier les valeurs dont les majuscules ont été transformées en minuscules :

```
select * from CLIENT order by lower(NOM) ;
```

Ceci fonctionne correctement, sauf pour les lettres accentuées, que la fonction SQL `lower` ne transforme pas.

Il est possible d'aborder le problème via la clause SQL `collate`, mais SQLfast offre une fonction qui résout plus simplement le problème dans la plupart des cas : `deaccent`, qui remplace chaque lettre accentuée par sa lettre de base : 'à' devient 'a' et 'À' devient 'A'. On écrira donc, de manière à regrouper en une même séquence toutes les variantes d'une même lettre :

```
select * from CLIENT order by lower(deaccent(NOM)) ;
```

De même, la requête suivante, qui recherche tous les livres dont le titre contient un mot déterminé, produira généralement des résultats incomplets (on notera l'usage de l'opérateur `§ §`) :

```
set mot = bûcheron;
select * from LIVRE
where TITRE like '%§mot§%';
```

La forme suivante nous donnera un résultat plus conforme à nos attentes :

```
set mot = bûcheron;
select * from LIVRE
where lower(deaccent(TITRE)) like lower(deaccent('%§mot§%')) ;
```

SQLfast offre la fonction `simplify`, qui combine les fonctions `lower` et `deaccent` :

```
where simplify(TITRE) like simplify('%§mot§%');
```

27. Affichage du résultat d'une requête

L'examen du contenu d'une table, d'une vue ou du résultat d'une requête s'affiche par défaut sur le **canal de sortie** courant, généralement la fenêtre de sortie. Le format standard est tabulaire avec encadrement des colonnes :

```
select NCLI,NOM from CLIENT where LOCALITE = 'Poitiers';
```

NCLI	NOM
B112	HANSENNE
F010	TOUSSAINT
C400	FERARD

28. Personnalisation du format d'affichage

Le format tabulaire dans lequel les instructions `select` affichent leur résultat est défini par 22 variables système, classées en sept type de lignes :

- un texte d'introduction (`selectIntro`),
- les 3 barres horizontales (`bar1`, `bar2`, `bar3`),
- la ligne de titre (`head`),
- les lignes de données (`data`),
- un texte de clôture (`selectClose`).

Chaque type de ligne, hormis `selectIntro` et `selectClose`, est défini par 4 chaînes de caractères selon sa position dans la ligne : 1 = extrême gauche, 2 = position d'un titre ou d'une valeur, 3 = séparation de titres/valeurs, 4 = extrême droite.

Illustration :

```

--> selectIntro
1----2---3-----2-----4 --> bar1 = (bar11,bar12,bar13,bar14)
1 222222 3 2222          4 --> head = (head1,head2,head3,head4)
1----2---3-----2-----4 --> bar2 = (bar21,bar22,bar23,bar24)
1 2222    3 22222222  4 --> data = (data1,data2,data3,data4)
| F010    | TOUSSAINT |
| C400    | FERARD    |
1----2---3-----2-----4 --> bar3 = (bar31,bar32,bar33,bar34)
--> selectClose
```

Le format standard est défini comme suit :

```

set bar11 = +-;      set bar21 = +-;      set bar31 = +-;
set bar12 = -;       set bar22 = -;       set bar32 = -;
set bar13 = +-;      set bar23 = +-;      set bar33 = +-;
set bar14 = -+@n;    set bar24 = -+@n;    set bar34 = -+@n;

set head1 = | ;      set data1 = | ;
set head2 = @h;      set data2 = @v;
```

```

set head3 = @s| ;      set data3 = @s| ;
set head4 = @s|@n;     set data4 = @s|@n;

set selectIntro = ;
set selectClose = ;

```

Une chaîne peut inclure des commandes de mise en forme (préfixe @) :

- @n : passer à la ligne
- @t : tabulation
- @h : à remplacer par le nom de la colonne
- @v : à remplacer par la valeur de la colonne dans la ligne de données
- @s : un espace
- @S5 : 5 espaces
- @A : insère la valeur courante du compteur auto-incrémenté A, défini par les deux variables systèmes selectA0 (valeur initiale) et selectAi (incrément). A est réinitialisé avant l'analyse de chaque ligne de données. Les variables selectA0 et selectAi sont initialisées avant affichage des données. Leurs valeur par défaut sont respectivement 0 et 1.
- @0, ..., @4 : à remplacer par la valeur actuelle des variables select0, ..., select4.
- @@ : remplacé par @.

Le paramètre `select-align` indique si les données de chaque ligne doivent suivre un alignement tabulaire (Y), comme dans le format standard, ou non (N), comme dans les formats `csv` ou `Latex` par exemple :

```
parameter select-align = Y;
```

La distribution SQLfast inclut treize formats prédéfinis codés sous la forme de scripts SQLfast. Le script du format désiré est à exécuter avant l'extraction des données. Penser à restaurer le format standard après l'extraction :

- format tabulaire standard
UTIL-SELECT-parameters-standard-1.sql
- format tabulaire standard simplifié
UTIL-SELECT-parameters-standard-2.sql
- format liste
UTIL-SELECT-parameters-for-listes.sql
- format tabulaire HTML
UTIL-SELECT-parameters-for-HTML.sql
- format clé-valeur
UTIL-SELECT-parameters-for-KEY-VALUE.sql
- format tabulaire Latex
UTIL-SELECT-parameters-for-LATEX.sql
- format tabulaire RTF (Microsoft)
UTIL-SELECT-parameters-for-RTF.sql

- instruction SQL insert
UTIL-SELECT-parameters-for-SQL.sql
- format tuples relationnels
UTIL-SELECT-parameters-for-TUPLE.sql
- format d'échange CSV (séparateur = ';')
UTIL-SELECT-parameters-for-CVS1.sql
- format d'échange CSV (séparateur = ',')
UTIL-SELECT-parameters-for-CVS2.sql
- format d'échange XML
UTIL-SELECT-parameters-for-XML.sql
- format d'échange JSON
UTIL-SELECT-parameters-for-JSON.sql

Exemple : génération de données au format *Clé-valeur* :

```
execSQL UTIL-SELECT-parameters-for-KEY-VALUE.sql;
select  NCLI,NOM,LOCALITE from CLIENT;
execSQL UTIL-SELECT-parameters-standard-1.sql
```

Ce qui donne :

```
NCLI = B112
NOM = HANSENNE
LOCALITE = Poitiers

NCLI = C123
NOM = MERCIER
LOCALITE = Namur

etc.
```

29. Supports d'écriture : les canaux de sortie

Les opérations qui écrivent des données (`write`, `select`, etc.) affichent celles-ci dans la *fenêtre de sortie*, à la suite des lignes déjà écrites. Il existe d'autres supports que cette fenêtre. Ces supports sont dénommés *canaux de sortie*. Le canal de sortie par défaut est la fenêtre de sortie. Un script peut selon ses besoins en choisir d'autres.

La sélection d'un canal s'effectue par les instructions `outputOpen` et `outputAppend`. L'instruction `outputOpen` ouvre le canal spécifié et efface son contenu précédent. L'instruction `outputAppend` ouvre le canal spécifié mais écrit à la suite du contenu précédent.

- Ecrire dans la fenêtre de sortie

```
outputOpen window;
outputAppend window;
```

- Ecrire dans un fichier de texte

```
outputOpen dépenses.csv;  
outputAppend dépenses.csv;
```

- Ecrire sur la console (équivalent à window dans la version actuelle)

```
outputOpen console;  
outputAppend console;
```

- Ecrire dans un fichier texte de même nom que le script principal

```
outputOpen standard;  
outputAppend standard;
```

- Ecrire dans une variable interne (le suffixe **.var** désigne la variable de resultat)

```
outputOpen resultat.var;  
outputAppend resultat.var;
```

L'ouverture d'un canal ferme d'abord le canal courant. Lorsqu'on a écrit dans un fichier, il est indispensable de le fermer en sélectionnant, par exemple, la fenêtre de sortie.

On peut désactiver/réactiver les opérations d'écriture par l'instruction `output` :

```
output off;  
output on;
```

A noter que l'instruction **print**, contrairement à **write**, affiche toujours la chaîne dans la fenêtre de sortie et **pas** sur le canal de sortie courant.

30. Les opérations sur les fichiers externes

SQLfast dispose de commandes permettant de créer, lire et gérer des fichiers externes.

a) Sélection d'un fichier ou d'un répertoire :

L'instruction de sélection d'un répertoire `selectDirectory` démarre dans un répertoire de référence (ici `D:/SQLfast`) et renvoie l'adresse complète du répertoire choisi (`source`) :

```
selectDirectory source = D:/SQLfast;  
write $source$;
```

L'instruction de sélection d'un fichier `selectFile` démarre dans un répertoire de référence (ici `D:/SQLfast/Scripts`), spécifie le type de fichier (SQL scripts) et son extension (`sql`). Elle renvoie l'adresse complète du fichier choisi (`code`) :

```
selectFile code = D:/SQLfast/Scripts, SQL scripts, sql;  
write $code$;
```

b) Manipulation de fichiers

La copie d'un fichier (`copyFile`) s'effectue sans erreur si le fichier source existe et si le fichier de destination n'existe pas. Dans les autres cas une erreur est déclenchée. Si le fichier de destination existe déjà, il est remplacé si le suffixe `-r` est spécifié (`copyFile-r`). Si le suffixe `-e` est spécifié (`copyFile-e`) aucune erreur n'est déclenchée si le fichier source est absent. Les deux suffixes peuvent être combinés.

```
copyFile D:/SQLfast/CLICOM.db, CLICOM-old.db;
```

L'instruction `renameFile` renomme un fichier. Une erreur est déclenchée si le fichier source n'existe pas ou si un fichier existe déjà sous le nouveau nom. Le suffixe `"-e"` supprime la notification dans le premier cas et le suffixe `"-r"` remplace le fichier existant dans le second cas. Ces deux suffixes peuvent être combinés.

```
renameFile D:/SQLfast/CLICOM.db, CLICOM-old.db;
```

L'instruction `deleteFile` supprime un fichier. La variante `deleteFile-e` ne déclenche pas d'erreur si le fichier n'existe pas.

```
deleteFile CLICOM.db;  
deleteFile-e CLICOM.db;
```

c) Consultation d'un répertoire

L'instruction `fileNames` explore un répertoire (`D:/SQLfast/Images`) et range dans un fichier de texte (`nomsF.txt`) la liste des noms de fichiers qui y sont localisés. Si une extension (`jpg`) est spécifiée, seuls les fichiers de cette extension sont repris. Les noms sont absolus, sauf si le suffixe `-s` est présent (`fileNames-s`), auquel cas seul les noms locaux sont enregistrés.

```
fileNames nomsF.txt = D:/SQLfast/Images, jpg;
```

La variante `fileNamesAll` explore de manière récursive l'arbre des répertoires au départ du répertoire de référence.

```
fileNamesAll nomsF.txt = D:/SQLfast/Images, jpg;
```

L'instruction `dirNames` explore un répertoire (`D:/SQLfast`) et range dans un fichier de texte (`nomsD.txt`) la liste des noms des répertoires qui y sont localisés. La variante `dirNamesAll` explore de manière récursive l'arbre des répertoires au départ du répertoire de référence. Le suffixe `-s` renvoie les noms locaux. Le répertoire de référence n'est pas inclus dans la liste sauf si le préfixe `-r` (`r` pour *root*) est spécifié.

```
dirNames nomsD.txt = D:/SQLfast;  
dirNamesAll nomsD.txt = D:/SQLfast;
```

Les instructions `dirFileNames` et `dirFileNamesAll` explorent un répertoire (D:/SQLfast) et rangent dans un fichier de texte (`nomsD.txt`) la liste des noms des répertoires et des fichiers qui y sont localisés. Les suffixes `-s` et `-r` sont admis.

```
dirFileNames nomsFD.txt = D:/SQLfast;
dirFileNamesAll nomsFD.txt = D:/SQLfast;
```

d) Transfert de/vers un fichier

L'instruction `variableToFile` copie le contenu d'une variable (`param`) dans un fichier (`parametres.txt`). Le fichier est créé s'il n'existe pas et est remplacé s'il existe. Le contenu de la variable est de nature quelconque.

```
variableToFile param = parametres.txt;
```

L'opération inverse est réalisée par l'instruction `fileToVariable` : copie du contenu d'un fichier dans une variable (`param`). Le contenu du fichier est un texte.

```
fileToVariable param = parametres.txt;
```

Si le contenu du fichier est de nature quelconque, on utilisera la variante `BfileToVariable`. Dans ce cas, la variable est du type *binaire* et ne peut plus apparaître dans un opérateur de substitution.

```
BfileToVariable musique = Files/Noel-en-Provence.mp3;
```

e) Lire et écrire dans un fichier de texte

La lecture des lignes successives d'un fichier de texte s'effectue par la boucle `for` sur fichier :

```
set fichier = Files/donnees.txt;
for ligne = [file $fichier] write $ligne$;
```

Il est possible également d'extraire les lignes successives via les primitives `openTextFile`, `nextTextLine` et `closeTextFile`.

```
openTextFile $fichier$;
while (True);
  nextTextLine ligne = $fichier$;
  if ('$FILEdiag$' = 'NONE') exit;
  write $ligne$;
endwhile;
closeTextFile $fichier$;
```

L'écriture dans un fichier consiste à choisir ce fichier comme canal de sortie par les instructions `outputOpen` ou `outputAppend`. Il est important de fermer ce fichier par le changement de canal de sortie avant de l'exploiter.

```
outputOpen Nombres.txt;
for I = [1,10] write $I$;
```

```
outputOpen window;  
fileToVariable liste = $outputDirectory$/Nombres.txt;  
write $liste$;
```

La dénomination des fichiers mérite un mot d'explication. Les instructions `outputOpen` et `outputAppend` qui spécifient un fichier par son *nom relatif* localisent celui-ci dans le répertoire par défaut `SQLfast/Output`. L'instruction `outputOpen Nombres.txt` crée donc, dans une installation standard, un fichier de nom `D:/SQLfast/Output/Nombres.txt`.

En revanche, l'instruction `fileToVariable` ne fait pas d'hypothèse sur la localisation du fichier source. Si le nom de ce dernier est local, le répertoire par défaut est le répertoire général `SQLfast`. Il est donc nécessaire de préciser le nom complet, sous la forme `$outputDirectory$/Nombres.txt`, qui, par substitution, donne `D:/SQLfast/Output.Nombres.txt`. CQFD.

Dans certains environnements, le caractère de fin de ligne du fichier n'est pas conforme à celui du nouveau canal de sortie. Dans ce cas, on réalisera son remplacement avant affichage par l'instruction `replace` :

```
replace liste = \n,@n;
```

31. Manipulation d'images

La visualisation d'une image s'effectue par l'instruction `showPicture`. Celle-ci spécifie une image **gif** ou **jpeg** ainsi que le message (optionnel) qui accompagne l'image:

```
showPicture Images/JeuxDePlage.gif = [/bJeux de plage];
```



L'image est fournie soit par un fichier d'extension **gif**, **jpg** ou **jpeg**, soit comme contenu d'une variable :

```
BfileToVariable var = Images/JeuxDePlage.gif;  
showPicture var = [/bJeux de plage];
```

L'instruction **BfileToVariable** est une variante de **fileToVariable** capable d'assigner à une variable SQLfast une valeur binaire non imprimable (image, son, document formaté, etc.) Tant qu'elle contiendra cette valeur, cette variable sera ignorée par les opérateurs de substitution.

Il est possible d'enregistrer et d'extraire des images d'une base de données. L'enregistrement se fait en deux temps. D'abord une ligne d'accueil est insérée, ensuite, on insère l'image dans cette ligne via une instruction **insertLOB** (**LOB** pour *Large Object*).

L'extraction de l'image suit la procédure habituelle.

Exemple, certes un peu alambiqué, mais à vocation didactique, d'une procédure de copie de fichier d'image :

```
createOrReplaceDB IMAGES.db;  
create table ALBUM(ImID integer not null primary key,  
IMAGE BLOB);
```

```
-- Insertion de la ligne 123
insert into ALBUM(ImID) values(123);
-- Lecture d'une image sur disque
BfileToVariable var1 = Images/JeuxDePlage.gif;
-- Insertion de l'image dans la ligne 123
insertLOB var1 = IMAGE from ALBUM where ImID = 123;
-- Extraction et copie sur disque de l'image 123
extract var2 = select IMAGE from ALBUM where ImID = 123;
variableToFile var2 = Images/JeuxDePlage-2.gif;
closeDB;
```

Remarques

La procédure d'enregistrement et d'extraction décrite ci-dessus vaut pour tout objet volumineux de nature généralement binaire : image, son, video, pdf, document avec mise en page.

Une table contenant des colonnes du type BLOB peut faire l'objet d'une requête select pour autant que ces colonnes ne contiennent que des caractères ou que la requête ne mentionne pas ces colonnes.

Dans les mêmes conditions, la table peut faire l'objet d'une opération Show schema mais pas Show data.

32. Gestion des erreurs

Lorsqu'une erreur survient lors de l'exécution d'un script, SQLfast affiche un message décrivant le problème. En outre, si le script dans lequel l'erreur s'est produite est présent dans la fenêtre principale, la ligne litigieuse est colorée en rouge. La description d'une erreur apparaît à la fois dans une fenêtre pop-up et sur le canal de sortie. Elle contient une trace complète de l'événement :

- la nature de l'erreur
- le nom complet du script dans lequel l'erreur s'est produite
- le numéro de la ligne ayant provoqué l'erreur.

En outre, si l'erreur s'est produite dans un script secondaire, pour chaque script appelant,

- le nom complet du script appelant
- le numéro de la ligne ayant appelé le script secondaire.

Lors de leur exécution, certaines instructions SQLfast renvoient, via des variables systèmes, des codes informant sur les conditions de l'exécution.

Pour les instructions relatives aux opérations sur la base de données, deux variables sont disponibles :

- **SQLdiag** : **OK** (pas d'erreur), **NONE** (objet non trouvé), **SYNTAX**, **DB** (erreur d'ouverture, fermeture ou absence de BD), **NULL** (violation *not null*), **VALUE** (valeur invalide), **ID** (violation contrainte d'unicité), **FK** (violation contrainte référentielle), **SCHEMA** (table ou colonne inconnue), **SYSTEM** (erreur SGBD), **ERROR** (autre erreur).

- **EXTENDEDdiag** : message d'erreur renvoyé par le SGBD et dépendant de celui-ci; donne plus de détails sur l'erreur.

Pour les instructions relatives aux opérations sur un fichier, une variable est consultable :

- **FILEdiag** : **OK** (pas d'erreur), **NONE** (fichier ou ligne non trouvé), **ERROR** (autre erreur).

L'instruction `onError` spécifie le comportement du moteur SQLfast lors de la survenance d'une erreur : soit stopper l'exécution du script et afficher un message d'erreur (comportement par défaut), soit poursuivre l'exécution sans avertissement :

```
onError stop;  
onError continue;
```

Il est possible, par l'instruction `error`, de créer artificiellement une erreur en mentionnant le message décrivant celle-ci. Le séquence suivante illustre son usage :

```
onError continue;  
error ER0503 Client inconnu;  
write $EXTENDEDdiag$;  
onError stop;  
error ER0503 Client inconnu;
```

Remarque

Dans la version actuelle de SQLfast (1.x), seules les instructions relatives aux *fichiers* et aux *bases de données* renvoient des codes d'erreurs. Le mode `onError continue` doit donc être utilisé avec prudence. Dans les versions ultérieures, toutes les instructions seront accompagnées de codes d'erreur spécifiques.

33. Arrêt d'un script

Supposons que le script **A**, lors de son exécution, appelle le script **B**, qui lui-même appelle le script **C**. Le script **A** est le script principal (lancé par le bouton **Run script** par exemple), **B** et **C** étant secondaires.

L'exécution du script **C** peut se terminer de différentes manières selon l'instruction qui s'y exécute :

- **return** : l'exécution de **C** se termine mais l'exécution du script appelant (ici **B**) se poursuit.
- **stop** : l'exécution de tous les scripts (**C**, **B** et **A**) se terminent; retour à la fenêtre principale.
- **shutdown** : l'exécution de tous les scripts (**C**, **B** et **A**) et de SQLfast lui-même se termine.

Dans un script principal, `return` et `stop` ont le même effet.

```
if ('$SQLdiag$' = 'NONE') return;
if ($errorCode$ = 100) stop;
shutdown;
```

Remarque

Un script est dit *principal* lorsqu'il est lancé à partir de la fenêtre principale par un des boutons **Run** ou exécuté en mode autoexec. Un script exécuté à la demande d'un autre script est dit *script secondaire*. *Principal* et *secondaire* ne sont pas des propriétés d'un script mais de sa position dans la chaîne d'exécution.

34. Contrôle de l'opérateur de substitution

Les opérateurs de substitution `$var$` et `%var%` agissent dans le prétraitement d'une instruction, avant son exécution. Ils spécifient le nom d'une variable qu'ils remplacent par la valeur de cette variable. Si la variable `var` n'existe pas, les expressions `$var$` et `%var%` restent inchangées. Un tel cas n'est donc pas une erreur.

L'opérateur standard `$var$` est spécifié par les délimiteurs par défaut `$` et `%`. Ceux-ci peuvent être modifiés dans un script par l'instruction :

```
parameter delimiters = [ ];
```

Seul le premier est obligatoire. Le retour aux délimiteurs par défaut est obtenu par :

```
parameter delimiters = default;
```

Les délimiteurs par défaut sont en fait spécifiés dans le fichier d'initialisation `SQLfast.ini` et peuvent également être modifiés :

```
delimiters = $ %
```

L'instruction `set` admet une variante, `set -n`, qui désactive le processus de substitution.

```
set -n A = %B%;
```

Ce mécanisme de substitution est parfois indésirable, en particulier dans les scripts qui génèrent d'autres scripts. Il est possible de le désactiver pour les instructions suivantes :

```
substitution off;
```

On le réactive par :

```
substitution on;
```

Ce qu'illustre le fragment ci-dessous, qu'on exécutera par curiosité :

```

set A = 123;
set-n B = $A$;
write B = $B$;
substitution off;
write B = $B$;
substitution on;
write B = $B$;

```

On peut également empêcher la substitution en supprimant au préalable la variable :

```

dropVariable var;
set A = $var$;

```

Cette technique sera utile dans un script **A** qui génère un script **B** qui utilise la variable `var`, laquelle n'est pas utilisée dans **A**, puis l'exécute.

Les délimiteurs § §

La variante `§var§` a pour effet de remplacer le nom de la variable par sa valeur, mais après avoir doublé les apostrophes que cette valeur contient. Ceci permet d'introduire des chaînes de caractères dans des instructions SQL, des expressions de calcul (`compute`) et des conditions (`if`, `while`), de manière à produire des instructions conformes à la syntaxe SQL.

Deux différences par rapport à l'opérateur `§ §` : (1) les délimiteurs de `§var§` ne sont pas modifiables, (2) cette forme ne fait pas l'objet de substitutions récursives comme `§ §`. Dès que la substitution `§ §` a été effectuée, la valeur introduite n'est plus réexaminée.

35. Contrôle de l'état d'un script

Un script dispose de plusieurs informations sur son état :

- **procLevel** : variable qui indique le niveau du script dans la chaîne d'appels. Le script principal est au niveau 0, le script qu'il appelle directement est niveau 1, et ainsi de suite. Comme un même script peut apparaître plus d'une fois dans la chaîne (lors d'appels récursifs), on parlera plutôt du processus en cours, c'est à dire de l'instance de script en cours d'exécution. Le niveau est limité à une valeur définie par le paramètre `maxProcLevel` dans le fichier d'initialisation `SQLfast.ini`. Sa valeur par défaut est 50.
- **scriptName** et **scriptNameFull** : nom local et nom complet du script en cours d'exécution.
- **mainScriptName** et **mainScriptNameFull** : nom local et nom complet du script principal à la source du script en cours d'exécution.

Les variables suivantes spécifient les ressources de l'environnement et leur localisation. Elles sont initialisées dans le fichier d'initialisation `SQLfast.ini`. Elles peuvent être consultées par les scripts et même être modifiées, mais ceci réclame la plus grande prudence !

- **DBMS** : SGBD utilisé par SQLfast.
- **platform** : système d'exploitation sous lequel tourne SQLfast.
- **sysPath** : répertoire dans lequel est localisé le logiciel SQLfast.
- **generalDirectory** : répertoire général de SQLfast. Est désigné dans la suite par le nom symbolique **_general**. Exemple : D:/SQLfast. Par défaut, est égal à sysPath.
- **scriptDirectory** : répertoire par défaut dans lequel sont localisés les scripts. Si son nom est local, il est défini dans le répertoire général. S'il est égal à **_general**, il est le répertoire général. Est désigné dans la suite par le nom symbolique **_script**. Exemple : **scriptDirectory** = **Scripts** (soit le répertoire D:/SQLfast/Scripts).
- **fileDirectory** : répertoire par défaut dans lequel sont localisés les fichiers produits et utilisés par les scripts. Si son nom est local, il est défini dans le répertoire général. S'il est égal à **_general** ou **_script**, il est le répertoire général ou le répertoire des scripts. Est désigné dans la suite par le nom symbolique **_file**.
- **dbDirectory** : répertoire par défaut dans lequel sont localisées les bases de données. Mêmes règles que ci-dessus. Exemple : **fileDirectory** = **Databases** (soit le répertoire D:/SQLfast/Databases).
- **helpDirectory** : répertoire par défaut dans lequel sont localisés les fichiers Help, les tutoriels et leurs figures. Mêmes règles que ci-dessus. Exemple : **helpDirectory** = **SQLfastHelp_FR** (soit le répertoire D:/SQLfast/SQLfastHelp_FR).

La manière dont les opérations sur une base de données et un fichier se terminent est décrite par les variables suivantes (voir rubrique *Gestion des erreurs*) :

- **SQLdiag** : code synthétique relatif à la base de données
- **EXTENDEDdiag** : code détaillé relatif à la base de données
- **FILEdiag** : code synthétique relatif aux fichiers.

La variable **DIALOGbutton** donne le nom du bouton qui a été utilisé pour fermer une boîte de dialogue.

Des instructions permettent d'afficher la liste des variables et leurs valeurs actuelles :

- **writeVariables**; : liste des variables utilisateur, créées par les scripts
- **writeSystemVariables**; : liste des variables système
- **writeParameters**; : liste des paramètres.

Toutes les variables sont globales et sont accessibles et modifiables par tous les scripts. En cas de besoin, il est possible de supprimer des variables :

- **dropVariable A**; : supprimer la variable utilisateur **A**
- **dropAllVariables**; : supprimer toutes les variables utilisateur

36. Exécution automatique d'un script

Le paramètre `autoexec` du fichier d'initialisation `SQLfast.ini` spécifie un script à exécuter automatiquement dès le démarrage de `SQLfast`.

```
autoexec = gestionBiblio.sql
```

Dès que ce script est terminé, la fenêtre principale est à nouveau disponible pour permettre à l'utilisateur de lancer l'exécution d'autres scripts. Cependant, si le script se termine par l'instruction.

```
shutdown;
```

alors `SQLfast` se termine également.

37. Exécution de scripts de grande taille

Il s'agit d'une alternative rapide à `execSQL`. Les instructions sont directement soumises au moteur SQL sans validation ni substitution de variables. Le script n'étant pas chargé en mémoire, il n'est pas soumis à une limitation de taille.

Le fichier source ne contient que des instructions SQL `create`, `insert`, `update` et `delete`. Les lignes blanches et les commentaires (--) sont admis.

L'instruction spécifie le fichier source et trois entiers : `first`, `last` et `step`. `SQLfast` exécute les instructions dont le rang dans le fichier source est compris dans l'intervalle `[first,last]`. Le paramètre `step` indique l'endroit dans l'intervalle auquel un checkpoint est exécuté. A chaque checkpoint, un `commitDB` est exécuté si le suffixe `'-c'` est spécifié. En outre, si le suffixe `'-t'` est spécifié, un message précisant le temps d'exécution est affichée.

```
fastExec      source.sql, 1000000, 2000000, 50000;  
fastExec-c    source.sql, 1000000, 2000000, 50000;  
fastExec-t    source.sql, 1000000, 2000000, 50000;  
fastExec-ct  source.sql, 1000000, 2000000, 50000;
```

En cas d'erreur, l'exécution s'arrête et renvoie le diagnostic dans `FILEdiag` ou `SQLdiag` selon la nature de l'erreur. `EXTENDEDdiag` indique le numéro de la ligne du fichier qui a provoqué l'erreur.

38. Génération-exécution de scripts

Un script peut générer un fichier contenant un script `SQLfast` puis en demander l'exécution. Les applications de ce modèle sont très nombreuses, en particulier pour spécialiser des scripts pour la base de données courante. Cette technique, très puissante, réclame un peu d'attention.

Examinons l'exemple ci-dessous, dans lequel le script crée un fichier contenant un petit script de 3 lignes, puis en demande l'exécution :

```
outputOpen $scriptDirectory$/scriptGenere.sql;  
  write openDB CLICOM.db;;  
  write select * from CLIENT;;  
  write closeDB;;  
outputAppend window;  
execSQL scriptGenere.sql;  
deleteFile $scriptDirectory$/scriptGenere.sql;
```

Commentaires

- outputOpen. L'instruction outputOpen dérive les écritures vers le fichier scriptGenere.sql, qui sera placé dans le répertoire par défaut \$scriptDirectory\$.
- write. Les trois instructions write créent les trois instructions du script. Attention à y inclure le signe ';' de fin d'instruction (il en faut donc deux).
- outputAppend. L'instruction outputAppend window est nécessaire pour reprendre la main mais aussi pour fermer le fichier et donc le libérer. Sinon, son exécution sera impossible (erreur Windows de code 32).
- execSQL. L'instruction execSQL exécute le nouveau script.
- deleteFile. L'instruction deleteFile (facultative) supprime ce script après exécution. Attention, il est nécessaire de préciser le répertoire de celui-ci (\$scriptDirectory\$/scriptGenere.sql) puisque l'instruction deleteFile ignore qu'il s'agit d'un script.

Si le script générateur et le script généré contiennent tous les deux une expression telle que \$var\$, une deuxième exécution du générateur risque d'échouer parce que la variable var, inconnue lors de la première génération, a été créée dans le script généré et possède désormais une valeur. Il sera prudent d'insérer au début du générateur une instruction "dropVariable var" qui rendra cette variable inconnue.

39. Les tables du dictionnaire de la base de données

Le dictionnaire SQLfast décrit le schéma de la base de données ouverte. Il est constitué de tables décrivant notamment les tables (SYS_TABLE), les colonnes (SYS_COLUMN), les clés (SYS_KEY) et les composants des clés (SYS_KEY_COMP). Il comprend également une vue (SYS_INTER_TABLE) qui décrit le graphe des tables et des clés étrangères.

Le dictionnaire est créé soit à l'ouverture de la base de données (selon la valeur du paramètre **Dictionary**, on ou off, du fichier d'initialisation SQLfast.ini), soit à la demande, via l'instruction

```
createDictionary;
```

Le dictionnaire est supprimé par l'instruction

```
deleteDictionary;
```

Les tables du dictionnaire sont temporaires et disparaissent lors de la fermeture de la base de données. Elles sont automatiquement mises à jour lors de l'exécution de requêtes DDL.

```
openDB CLICOM.db;
createDictionary;
select TableID,TableName from SYS_TABLE;
closeDB;
```

La structure et le contenu des quatre tables mentionnées sont consultables à l'aide du petit script :

```
openDB CLICOM.db;
createDictionary;
write-b Dictionary table SYS_TABLE;
select * from SYS_TABLE;
write-b Dictionary table SYS_COLUMN;
select * from SYS_COLUMN;
write-b Dictionary table SYS_KEY;
select * from SYS_KEY;
write-b Dictionary table SYS_KEY_COMP;
select * from SYS_KEY_COMP;
write-b Dictionary view SYS_INTER_TABLE;
select * from SYS_INTER_TABLE;
closeDB;
```

Une version simplifiée, mais suffisante pour rédiger des requêtes SQL, est disponible via le menu **Database > Show schema** de la fenêtre principale. Ce schéma s'affiche, pour la base de données sélectionnée, dans une *fenêtre de schéma*.

On rappelle qu'un schéma plus complet peut être obtenu en cliquant sur l'étiquette Summary à gauche du bandeau supérieur et en sélectionnant le format Full schema. Dans ce cas, c'est le contenu du fichier CLICOM.ddl qui apparaît dans la fenêtre.

Si ce fichier n'existe pas, il peut être créé automatiquement par le script SQL-DDL-Generation.sql disponible dans le répertoire par défaut des scripts.

Summary schema		Full schema	Close
Database: D:/SQLfast/Databases/CLICOM.db			
Tables and views of database "CLICOM.db"			
table CLIENT			
* NCLI	char(10)	PK1	
* NOM	char(32)		
* ADRESSE	char(60)		
* LOCALITE	char(30)		
CAT	char(2)		
* COMPTE	decimal(9,2)		
table PRODUIT			
* NPRO	char(15)	PK1	
* LIBELLE	char(60)		
* PRIX	decimal(6)		
* QSTOCK	decimal(8)		
table COMMANDE			
* NCOM	char(12)	PK1	
* NCLI	char(10)		
* DATECOM	date		
table DETAIL			
* NCOM	char(12)	PK1	
* NPRO	char(15)	PK2	
* QCOM	decimal(8)		

40. Visualisation des données en HTML

Important :

Vérifier que la variable système HTMLprocessor a bien été initialisée dans le fichier SQLfast.ini avec le nom complet du navigateur.

Générer une page HTML à partir du contenu d'une base de données est assez simple. Produire le code HTML de présentation tabulaire de données extraites l'est encore plus, grâce au format défini par le script UTIL-SELECT-parameters-for-HTML.sql.

Le script ci-dessous range dans le fichier HTML.html le code HTML désiré, ferme ce fichier, affiche son contenu à l'aide du navigateur par défaut dont l'adresse est définie par la variable système HTMLprocessor dans le fichier de paramètres SQLfast.ini.

```
openDB CLICOM.db;
outputOpen HTML.html;
execSQL UTIL-SELECT-parameters-for-HTML.sql;
select * from CLIENT where LOCALITE = 'Toulouse';
execSQL UTIL-SELECT-parameters-standard-1.sql;
outputAppend window;
execProg $HTMLprocessor$ $outputDirectory$/HTML.html;
closeDB;
```

Dans la version plus générale ci-dessous, le nom de la localité est demandé à l'utilisateur et le fichier HTML est supprimé après affichage.

Précaution indispensable : une temporisation d'une seconde (`wait 1000`) est imposée au script avant la suppression du fichier de manière à laisser au navigateur le temps d'afficher celui-ci.

Remarque : si SQLfast sait où écrire les fichiers produits (par défaut, le répertoire Output), il n'en est pas de même du navigateur ni de l'opération de suppression. D'où l'usage de l'adresse absolue du fichier HTML `$outputDirectory$/HTML.html`.

```
openDB CLICOM.db;
askData loc = Nom de localité:
[select distinct LOCALITE from CLIENT];
set locFile = $outputDirectory$/HTML.html;
outputOpen $locFile$;
execSQL UTIL-SELECT-parameters-for-HTML.sql;
select * from CLIENT where LOCALITE = '$loc$';
execSQL UTIL-SELECT-parameters-standard-1.sql;
outputOpen window;
execProg $HTMLprocessor$ $locFile$;
wait 1000;
deleteFile $locFile$;
closeDB;
```

NCLI	NOM	ADRESSE	LOCALITE	CAT	COMPTE
B512	GILLET	14, r. de l'Eté	Toulouse	B1	-8700
C003	AVRON	8, ch. de la Cure	Toulouse	B1	-1700
K729	NEUMAN	40, r. Bransart	Toulouse	--	0
F011	PONCELET	17, Clô des Erables	Toulouse	B2	0
D063	MERCIER	201, bvd du Nord	Toulouse	--	-2250

41. Ajout d'un mode d'emploi à une application SQLfast

Les fichiers d'aide et les tutoriels sont affichés dans une fenêtre de tutoriel via le bouton **Help** du niveau *Learning SQL* et via le menu **Help** du niveau *Learning SQLfast*.

Le langage SQLfast dispose de deux instructions qui permettent d'ouvrir et de fermer un mode d'emploi, un tutoriel ou un fichier d'aide au départ d'un script.

Le fragment ci-dessous, supposé appartenir à un script dédié à l'apprentissage du langage SQL, ouvre une fenêtre d'aide et y charge le fichier `1_Requetes_elementaires.tuto`, lance le script d'apprentissage `Module-SQL-01.sql`, puis ferme la fenêtre d'aide après avoir affiché un message de clôture :

```
displayHelp Lev0_Tutorials/1._Requetes_elementaires.tuto;
set score = 0;
execSQL Module-SQL-01.sql;
```

```

showMessage La leçon est terminée.
                @nVous avez obtenu un score de $score$.;
closeHelp;

```

42. Extraction de données d'un site web

Extraire des données et des documents d'un site web est assez simple lorsque le volume à traiter est très réduit : on affiche les données dans la fenêtre d'un navigateur, on en extrait des fragments de texte désirés qu'on copie dans le presse-papier, puis qu'on colle dans un document local.

Pour les images, les clips vidéo et les documents plus complexes (zip ou pdf par exemple), on utilisera la fonction de téléchargement du navigateur (typiquement: clic droit, puis *Enregistrer la cible du lien*).

Le problème est beaucoup plus complexe dès que le nombre et le volume de données devient plus important.

Un exemple

Supposons que nous désirions constituer une base de données sur la faune européenne. Nous avons identifié un site de qualité qui fournit les caractéristiques de plusieurs milliers d'espèces, accompagnées chacune d'une photo. Supposons aussi que le propriétaire de ce site n'est pas disposé à nous livrer sa propre base de données mais qu'il ne s'oppose pas à ce que nous extrayions les données de ses pages web.

Le site source est constitué de pages présentant chacune dix fiches individuelles. L'examen du code HTML d'une page nous apprend que :

- l'adresse (url) d'une page de fiche est formée d'un préfixe commun suivi d'un numéro de séquence : par exemple, "http://www.animaux.europe.org/fiches/page_218.html" pour la 218e page.
- dans chaque page, chaque fiche est traduite sous la forme d'une division encadrée des deux balises "<div class='fichephp'>" et "".
- la division d'une fiche comporte une suite de chaînes détaillant les propriétés d'une espèce : *nom commun*, *nom scientifique*, *famille*, *biotope*, *taille*, etc. Chaque propriété est représentée par une division telle que : "<div>Famille:bovidé</div>".
- la division de chaque fiche contient également l'adresse d'une image comprise entre les deux chaînes "<a href='" et ".jpg".

Le script

Le script principal calcule la suite des url des pages, ici limitées à 1000. Pour chacune, le code HTML est téléchargé et stocké dans la variable `page`. Il y est ensuite analysé par la procédure `Analyser-page.sql`.

```

for I = [1,1000];
  set url = http://www.animaux.europe.org/fiches/
    page_${I}.html;
  webPageToVariable page = $url$;
  if ('$WEBdiag$' = 'NONE') exit;
  execSQL Analyser-page.sql;
endfor;

```

La procédure `Analyser-page.sql` extrait les sections successives de la page courante par la fonction `Extractsection` de la bibliothèque `LStr`. Chaque section, qui représente la fiche d'un animal, est ensuite exploitée comme suit :

- la procédure `Extraire-proprietes.sql` extrait les données de la fiche et l'adresse de la photo,
- les données extraites sont stockées dans la table `FICHE` de la base de données (instruction `insert`),
- la photo de la fiche est téléchargée puis stockée dans le répertoire `ImagesAnimaux`.

```

set pos = 0;
while (True);
  function fiche,pos Lstr:Nextsection $page$, $pos$,
    <div class="fichephp">,<div>,</div>;
  if ($pos$ = -1) exit;
  execSQL Extraire-proprietes.sql;
  insert into FICHE (NomCommun,NomScientifique,...,Image)
    values ($nom-commun$, $nom-scient$, ..., $urlImage$);
  webFileToFile $fileDirectory$/ImagesAnimaux
    = html$urlImage$.jpg;
endwhile;

```

La procédure `Extraire-proprietes.sql` recherche les valeurs des propriétés de la fiche à l'aide des balises identifiant chacune :

```

function nom-commun = Lstr:ExtractTaggedString fiche,
  <div>Nom commun:,</div>;
function nom-scient = Lstr:ExtractTaggedString fiche,
  <div>Nom scientifique:,</div>;
...
function urlImage = Lstr:ExtractTaggedString fiche,
  <a href=html',.jpg;

```

Commentaires

Moins d'une vingtaine d'instructions suffisent donc pour extraire les données et les images d'un site web et les stocker dans une base de données.

Pour exploiter ces données, il faut ensuite convertir les images `jpg` en format `gif` puis rédiger les scripts qui permettent de présenter, interroger et explorer les données.

Quelques mots sur les deux instructions permettant d'interagir avec le web :

- `webPageToVariable` : stocke dans une variable le code HTML de la page dont on spécifie l'url; certaines conversions facilitant l'analyse du code sont effectuées,

- `webFileToFile` : télécharge un fichier dont on donne l'url et le stocke sous un nom local, ou le range dans le répertoire spécifié. Si l'url désigne une page HTML, son code est stocké dans ce fichier.
- sur les deux fonctions de la bibliothèque `LStr` que nous avons utilisées :
- `nextSection texte,pos,arg1,arg2,arg3` : analyse la chaîne `texte` à partir de la position `pos`, y recherche le plus petit fragment encadré par les sous-chaînes `arg1` et `arg3` (une "section") tout en ignorant les sous-chaînes encadrées par `arg2` et `arg3`. Renvoie la section identifiée et la position du caractère qui suit `arg3` (`pos` vaut -1 si aucune section n'a été trouvée).
- `ExtractTaggedString texte,arg1,arg2` : renvoie le premier fragment de la chaîne `texte` compris entre les sous-chaînes `arg1` et `arg2`.

43. Données temporaires : table ou liste SQLfast ?

Le stockage temporaire d'une liste de valeurs se fait idéalement dans une table ... temporaire :

```
create temp table LOC as
select distinct LOCALITE from CLIENT order by LOCALITE;
```

On peut en effet les manipuler aisément à l'aide du langage SQL. Dans certains cas cependant, on peut trouver excessif de faire appel à toute la puissance de ce langage pour stocker et manipuler de simples listes. On utilisera alors les *listes SQLfast*.

a) Rappel : la liste SQLfast

Une liste SQLfast est une chaîne de caractères constituée d'une suite de valeurs séparées par un symbole dit *séparateur*. Ainsi, le contenu de la table LOC pourrait être plus simplement remplacé par la liste :

```
Bruxelles;Genève;Lille;Namur;Paris;Poitiers;Toulouse
```

Le séparateur est ici le point-virgule (`' ; '`). Il s'agit du séparateur standard par défaut mais toute chaîne peut être utilisée pour séparer les valeurs.

Les éléments d'une liste sont des chaînes de caractères, chaînes numériques lorsque ces éléments sont des nombres :

```
Paris;123;4.50;-68;1.25E-8
```

Pour manipuler une liste il faut connaître la valeur du séparateur. Celle-ci doit donc être spécifiée lors de l'utilisation des fonctions de listes. Pour obtenir le 3e élément de la liste `$L$`, on utilise la fonction **item**, pour laquelle on spécifie la liste (`' $L$ '`), la position de l'élément recherché (3) et le séparateur (`' ; '`) :

```
item('$L$',3,';')
```

Si le séparateur est vide, alors chaque caractère de la liste est un élément distinct. L'expression suivante renvoie la lettre 'c' :

```
item('abcdefghijklmnopqrstuvwxyz', 3, '')
```

Un élément peut être vide, tel que le 3e de la liste suivante :

```
Bruxelles;Genève;;Namur;Paris;Poitiers;Toulouse
```

Lorsqu'un élément comprend les caractères du séparateur, on le protégera par des apostrophes. Dans l'exemple ci-dessous, le séparateur est la virgule :

```
Bruxelles,Genève,Lille,Namur,'Paris, ville lumière',  
Poitiers, Toulouse
```

Plus généralement, tout élément PEUT être entouré d'apostrophes mais il DOIT l'être s'il inclut le séparateur. Si un élément protégé comprend des apostrophes, on traitera celles-ci comme il est d'usage dans les constantes SQL, en les doublant :

```
Bruxelles,Genève,Lille,Namur,Paris,Poitiers,  
'Villeneuve d'Ascq',Toulouse
```

b) Création d'une liste SQLfast

Comme toute valeur d'une variable, une liste sera créée par assignation (`set`), calcul (`compute`), lecture dans un fichier ou extraction d'une base de données (`extract`). Dans ce dernier cas, on utilisera la fonction agrégative `group_concat`, avec ou sans doublons, avec ou sans tri :

```
extract L = select group_concat2(LOCALITE,1,1,0,',')  
from CLIENT;
```

c) Manipulation de listes SQLfast (rappel)

Manipulations élémentaires

- Élément de rang donné d'une liste.

```
compute elem = item('$L$', 3, ',')
```

- Nombre d'éléments dans une liste, y compris les éléments vides.

```
compute long = itemLen('$L$', ',')
```

- Partie d'une liste comprenant les éléments de rang 3 à 5, ceux-ci compris.

```
compute sous-liste = itemSlice('$L$', 3, 5, ',')
```

- Position dans une liste du premier élément de valeur donnée.

```
compute position = itemIndex('$L$', 'Toulouse', ',')
```

- Tri des éléments d'une liste, chaque élément étant considéré comme une chaîne de caractères non numériques. Par exemple, '5' > '12'. Le paramètre D indique la direction du tri : 0 = ascendant, 1 = descendant. Le paramètre U indique si les doublons sont éliminés : 0 = non, 1 = oui.

```
compute triée = itemSort('$L$', D, U, ',')
```

- Tri des éléments d'une liste, chaque élément étant considéré comme une chaîne de caractères représentant un nombre. Par exemple, '5' < '12'. Paramètres D et U comme ci-dessus.

```
compute triée = itemNSort('$L$', D, U, ',')
```

- Supprimer l'élément de rang donné d'une liste.

```
compute réduite = itemDelete('$L$', 3, ',')
```

Jointure de listes

- Liste résultant de la concaténation des éléments de même rang de deux listes, séparés par le séparateur spécifié. Le 1er séparateur est celui des deux listes, le 2e sépare les deux valeurs sources et le 3e sépare les éléments du résultat.

Exemple : `itemJoin('ABCD', '1234', ',', '- ', ',')`
 = 'A-1,B-2,C-3,D-4'

```
compute L12 = itemJoin('$L1$', '$L2$', ', ', '; ', ',')
```

Manipulations ensemblistes de listes SQLfast

Il est possible de construire une nouvelle liste par des opérations ensemblistes appliquées à deux listes sources. On peut aussi tester si deux listes sont égales ou si l'une est un sous-ensemble d'une autre. Ces opérations existent en deux variantes : sans doublons (ensembles purs) et avec doublons (multi-ensembles). Certains de ces opérateurs sont semblables à ceux de SQL.

- Union ensembliste de deux listes. La liste résultante comprend tous les éléments des listes sources L1 et L2, mais en un seul exemplaire, doublons exclus. Similaire à l'opérateur SQL union.

```
compute L = itemUnion('$L1$', '$L2$', ',')
```

- Union multi-ensembliste de deux listes. La liste résultante comprend tous les éléments des listes sources L1 et L2, doublons compris. Similaire à l'opérateur SQL union all.

```
compute L = itemUnionAll('$L1$', '$L2$', ',')
```

- Différence ensembliste de deux listes. La liste résultante comprend les éléments de L1 qui ne sont pas dans L2, mais en seul exemplaire, doublons exclus. Similaire à l'opérateur SQL `except`.

```
compute L = itemExcept('$L1$', '$L2$', ' , ')
```

- Différence multi-ensembliste de deux listes. La liste résultante comprend les éléments de L1 qui ne sont pas dans L2, en tenant compte des doublons. Similaire à l'opérateur SQL `except all`.

```
compute L = itemExceptAll('$L1$', '$L2$', ' , ')
```

- Intersection ensembliste de deux listes. La liste résultante comprend les éléments de L1 qui sont aussi dans L2, mais en seul exemplaire, doublons exclus. Similaire à l'opérateur SQL `intersect`.

```
compute L = itemInter('$L1$', '$L2$', ' , ')
```

- Intersection multi-ensembliste de deux listes. La liste résultante comprend les éléments de L1 qui sont aussi dans L2, en tenant compte des doublons. Similaire à l'opérateur SQL `intersect all`.

```
compute L = itemInterAll('$L1$', '$L2$', ' , ')
```

- Equivalence ensembliste de deux listes. Indique si les listes L1 et L2 comprennent les mêmes éléments, doublons exclus. Il n'existe pas d'opérateur équivalent en SQL.

```
compute égales = itemEqual('$L1$', '$L2$', ' , ')
```

- Equivalence multi-ensembliste de deux listes. Indique si les listes L1 et L2 comprennent les mêmes éléments, doublons compris. Il n'existe pas d'opérateur équivalent en SQL.

```
compute égales = itemEqualAll('$L1$', '$L2$', ' , ')
```

- Inclusion ensembliste de deux listes. Indique si les éléments de la liste L1 appartiennent également à L2, doublons exclus. Il n'existe pas d'opérateur équivalent en SQL.

```
compute inclue = itemInclude('$L1$', '$L2$', ' , ')
```

- Inclusion multi-ensembliste de deux listes. Indique si les éléments de la liste L1 appartiennent également à L2, doublons compris. Il n'existe pas d'opérateur équivalent en SQL.

```
compute inclue = itemIncludeAll('$L1$', '$L2$', ' ', '')
```

- Traduction d'une chaîne de caractères. retourne la chaîne S dans laquelle chaque instance de chaque élément de la liste L1 a été remplacée par l'élément correspondant de la liste L2. Similaire à la fonction SQL `translate` mais qui fonctionne également pour la substitution de sous-chaînes. L'intersection des listes L1 and L2 doit être vide (sinon ...).

```
set S = Le petit génie de la famille;
set L1 = petit,génie;
set L2 = gros,vantard;
compute conv = itemTranslate('$$$', '$L1$', '$L2$', ' ', ' ');
compute conv = itemTranslate('meilleurs voeux',
                             'abcdefghijklmnopqrstuvwxy',
                             'OPQRSTUVWXYZABCDEFGHIJKLMN', '');
```

d) Boucle sur liste (version 3.0)

Il existe une forme particulière de boucle qui itère sur les éléments successifs d'une liste SQLfast. Si la liste utilise le séparateur par défaut (';' par exemple) on écrira :

```
for ville = [items Bruxelles;Genève;Lille;Namur]
write $ville$;
```

Si la liste est construite sur un séparateur spécifique (' ' par exemple), on le spécifiera comme suit :

```
for ville = [item(,) Bruxelles,Genève,Lille,Namur]
write $ville$;
```

e) Exemples d'application

Ce script demande à l'utilisateur de sélectionner, via une boîte `selectMany`, une ou plusieurs localités parmi celles des clients. La liste des clients de chaque localité sélectionnée est ensuite affichée. La structure de la boîte de dialogue dépend des données courantes : liste des localités (labels) et liste des variables (variables) qui vont enregistrer la sélection. Ces deux listes vont servir à constituer l'instruction `selectMany`. La boucle examine le contenu des variables cibles de cette instruction. Le nom de chaque variable est stocké dans la variable `ville`; donc `$ville$` est un nom de localité et `$$ville$$` est la valeur, 0 ou 1, assignée à cette localité par l'utilisateur.

```
openDB CLICOM.db;
extract variables,labels = select
                             group_concat2(LOCALITE,1,1,0,' '),
                             group_concat2(LOCALITE,1,1,0,'|')
                             from CLIENT;
selectMany $variables$ = $labels$;
if ('$DIALOGbutton$' = 'Cancel') stop;

compute nombre = itemLen('$variables$', ' ', '');
```

```

for v = [1,$nombre$];
  compute ville = item('$variables$', $v$, ',');
  if ($$ville$$ = 1) select NCLI,NOM,LOCALITE
                     from CLIENT
                     where LOCALITE = '$ville$';
endfor;

-- variante plus concise de la boucle "for" :
for ville = [item(,) $variables$]
  if ($$ville$$ = 1)
    select NCLI,NOM,ADRESSE,LOCALITE
    from CLIENT where LOCALITE = '$ville$';

```

Dans ce deuxième exemple, on applique la technique au concept de *liste de listes*. La liste principale contient une séquence d'éléments, qui, chacun, est à son tour une liste. A titre d'exemple, représentons une partie des données de la table CLIENT sous une telle forme :

```

B062,GOFFIN,'72,r. de la Gare',Namur;C400,FERARD,'65,r. du
Tertre',Poitiers

```

Pour exploiter cette liste, que supposons stockée dans la variable clients, nous construisons deux boucles emboîtées :

```

for client = [items $clients$]
  for attribut = [item(,) $client$]
    write $attribut$;

```

44. Tables sans schéma : les colonnes dynamiques (version 3.0)

La procédure habituelle de création d'une table utilise le sous-langage SQL-DDL :

```

create table ARTICLE(ArtID integer not null primary key,
                     Nom varchar(20) not null);

```

L'insertion des données doit être conforme aux colonnes qui constituent la table :

```

insert into ARTICLE(ArtID,Nom)
values (123,'VIS (45-2)', (124,'POINTES (120-3.5)');

```

Si une nouvelle colonne est nécessaire, elle doit être créée par une requête SQL-DDL qui modifie le schéma de la table :

```

alter table ARTICLE add column Couleur char(20);

```

Il est dès lors possible d'insérer des lignes selon ce nouveau format :

```

insert into ARTICLE(ArtID,Nom,Couleur)
values (138,'BROSSE AD8','BLEU');

```

Pour toutes les lignes préexistantes, la nouvelle colonne a une valeur "null".

a) Les colonnes dynamiques

Pour certains domaines d'application, cette discipline stricte est considérée comme excessivement rigide. Lorsque les faits à enregistrer dans une table ont des propriétés très variables et impossibles à identifier à l'avance, on ne peut plus construire le schéma de cette table a priori. En pratique, on voudrait que le schéma de la table évolue dynamiquement en fonction des données qu'on y insère. Cette souplesse d'évolution du schéma est l'objectif des *colonnes dynamiques*.

Ce type de colonnes permet d'écrire et d'exécuter le script suivant, dont les requêtes `insert`, `update` et `select` référencent des colonnes non déclarées :

```
create table ARTICLE(ArtID integer not null primary key,
                     NOM   varchar(20) not null);

insert into ARTICLE(ArtID,NOM,COULEUR,PRIX)
  values (138,'BROSSE AD8','BLEU',2.50),
         (150,'BOITE OUTILS','GRIS',18);

update ARTICLE
set    COULEUR = 'NOIR',
      PRIX = 10
where  COULEUR is null;

select ArtID,NOM,COULEUR,FOURNISSEUR
from   ARTICLE
where  COULEUR = 'BLEU';
```

Les colonnes `Couleur`, `Prix` et `Fournisseur` sont dites *colonnes dynamiques*. Elles sont automatiquement créées la première fois qu'elles sont mentionnées dans une requête SQL. Les colonnes créées par les instructions `create table` et `alter table` sont appelées *statiques*. Une table constituée d'une colonne statique déclarée `primary key` et dont toutes les autres colonnes sont dynamiques peut être considérée comme une *table sans schéma*.

b) Conditions d'utilisation

Les colonnes dynamiques de SQLfast sont implémentées par une couche créée au-dessus du moteur SQLite et non par une extension de celui-ci. Cette architecture impose quelques contraintes :

- Pour des raisons d'efficacité du prétraitement des requêtes, il est nécessaire de *marquer* les noms des colonnes dynamiques par un préfixe spécifique. Ainsi, on écrira `!Couleur` au lieu de `Couleur`. Le marqueur par défaut `!` peut être modifié : il est défini par la variable système `dynaMarker`.
- Les colonnes dynamiques sont prises en charge (prétraitées) si la variable système `dynaColumns` est égale à 1. Elles sont ignorées si cette variable est égale à 0, ce qui est sa valeur par défaut. Cette variable est assignée dans le fichier `SQLfast.ini` :

```
dynaColumns = 1
```

ou dans le script lui-même :

```
set dynaColumns = 1;
```

- SQLfast ne mémorise pas les métadonnées relatives aux colonnes dynamiques. Les références à une colonne dynamique dans une requête sont identifiées par le mot qui suit le marqueur '!' (ou son substitut). Cette technique peut entraîner des ambiguïtés si le marqueur apparaît également dans des constantes SQL. On veillera dans ce cas à remplacer le marqueur par un symbole plus approprié.

Le script ci-dessus (légèrement augmenté !) doit donc être écrit comme suit :

```
createDB InMemory;
set dynaColumns = 1;

create table ARTICLE(ArtID integer not null primary key,
                     NOM varchar(20) not null);

insert into ARTICLE(ArtID,NOM,!COULEUR,!PRIX)
      values (138,'BROSSE AD8','BLEU',2.50);
insert into ARTICLE(ArtID,NOM,!COULEUR,!PRIX)
      values (150,'BOITE OUTILS','GRIS',18);

select ArtID,NOM,!COULEUR,!PRIX from ARTICLE;

update ARTICLE
set    !COULEUR = 'NOIR',
      !TAILLE  = 'MEDIUM'
where  !COULEUR = 'BLEU';

select ArtID,NOM,!COULEUR,!PRIX,!TAILLE from ARTICLE;

select ArtID,NOM,!COULEUR,!FOURNISSEUR
from   ARTICLE
where  !COULEUR = 'NOIR';
```

La syntaxe des clauses `set` de la requête `update` est soumise à la contrainte suivante :

la partie droite de chaque clause est soit une constante caractères dont la première position est une apostrophe, soit une constante numérique, soit une expression entre parenthèses.

On écrira donc :

```
update ARTICLE
set    !COULEUR = 'NOIR',
      !MODELE  = 12.5,
      NOM      = (lower(NOM) || '!'),
      !PRIX    = (!PRIX * 1.05)
where  !COULEUR = 'BLEU';
```

c) Métadonnées des colonnes dynamiques

La liste des noms des colonnes dynamiques d'une ligne s'obtient par la fonction (UDF) SQL `getDColNames(DColumns,sep)`, où `sep` est le séparateur de noms. Suite à l'exécution du script ci-dessus, la requête :

```
select getDColNames(DColumns,sep) as Colonnes from ARTICLE
where ArtID = 150;
```

produira le résultat :

```
+-----+
| Colonnes |
+-----+
| COULEUR,PRIX |
+-----+
```

Pour obtenir la liste des noms de TOUTES les colonnes dynamiques d'une table, on utilisera la fonction agrégative `group_DColNames(DColumns,sep)`:

```
select group_DColNames(DColumns,',') as Colonnes
from ARTICLE;
```

dont le résultat sera, dans les mêmes conditions :

```
+-----+
| Colonnes |
+-----+
| COULEUR,PRIX,TAILLE |
+-----+
```

Une petite application vient immédiatement à l'esprit : lister le contenu d'une table en y incluant les colonnes dynamiques. C'est ce que fait le script ci-dessous :

```
createDictionary;
extract listCol0 = select group_concat(ColName)
                    from SYS_COLUMN C, SYS_TABLE T
                    where C.TableID = T.TableID
                    and T.TableName = 'ARTICLE' and
C.ColName <> 'DColumns';
extract listeCol1 = select group_DColNames(',') from ARTICLE;
compute listeCol2 =
itemJoin('!'||replace('$listeCol1$',',',' ','!'),
'$listeCol1$',',',' ' as ',',,');
select $listCol0$, $listeCol2$ from ARTICLE;;
```

- La variable "listCol0" contient la liste des colonnes statiques : 'ArtID,NOM'.
- La variable "listCol1" contient la liste des colonnes dynamiques : 'COULEUR,PRIX,TAILLE'.
- La variable "listCol2" contient la liste des colonnes dynamiques avec leurs alias: '!COULEUR as COULEUR,!PRIX as PRIX,!TAILLE as TAILLE'.

Résultat de ce script, appliqué à la base de données dans l'état ci-dessus :

Résultat de ce script :

ArtID	NOM	COULEUR	PRIX	TAILLE
138	BROSSE AD8	NOIR	2.5	MEDIUM
150	BOITE OUTILS	GRIS	18	--

d) Note technique

Les formules d'extraction des métadonnées des colonnes dynamiques méritent un mot d'explication quant à leur implémentation. Dès qu'une colonne dynamique est mentionnée dans une requête `insert` ou `update`, une colonne technique nommée **DColumns** est ajoutée à la table par une requête `alter table ARTICLE add column ...`. Cette colonne contient un objet JSON dont les composants représentent les valeurs des colonnes dynamiques. Par exemple, la valeur de la colonne **DColumns** de la ligne `ArtID = 123` est exprimée comme suit :

```
{ "COULEUR": "NOIR", "PRIX": 10 }
```

Ceci explique la référence à cette colonne dans les fonctions `getDColNames(DColumns,sep)` et `group_DColNames(DColumns,sep)`.

D'autre part, on évitera d'utiliser la forme `"select * from ARTICLE"`, qui ferait apparaître la colonne technique au lieu des colonnes dynamiques.

