Ch. 2. Problèmes liés au SQL

De Apache OpenOffice Wiki
< FR | Documentation | HSQLDB Guide

Test de

Sommaire

l'extension SUN WIKI PUBLISHER 1.0 oxt. Menu Fichier > Exporter > Format de fichier

(Le document original, à valeur normative, est à cette page : http://hsqldb.org /doc/guide/ch02.html)

Fred Toussi

Groupe de Développement HSQLDB

<ft@cluedup.com (mailto:ft@cluedup.com)>

Note de traduction : Ici, il y a néanmoins l'altération de la traduction. En cas de doute, se référer au document original, à valeur normative, à l'adresse : http://hsqldb.org/web/hsqlDocsFrame.html

$Date: 2005/07/01 17:06:32 $

But de ce document

Beaucoup de questions fréquemment posées dans les Forums et Mailing-Lists trouvent leur réponse dans ce guide. Si vous voulez utiliser HSQLDB avec votre application, vous devez lire ce guide.

Support du standard SQL

HSQLDB 1.8.0 supporte le langage SQL défini par les standards SQL 92, 99 et 2003. Cela signifie que quand une caractéristique de ce standard est supportée, ex.: left outer join, la syntaxe est celle qui est spécifiée par le texte standard. Nombre de caractéristiques du SQL 92 et 99 sont supportées jusqu'à un niveau avancé, il y a un support pour la plupart des mots de la SQL 2003 Foundation et pour plusieurs fonctionnalités optionnelles de ce standard. Toutefois, certaines fonctionnalités ne sont pas supportées, aussi nous ne prétendons pas assurer un support pour la totalité des caractéristiques des standards.

Le chapitre Syntaxe SQL de ce guide liste tous les mots clés et la syntaxe utilisée. Lors de l'écriture ou de la conversion de la grammaire SQL DDL (Data Definition Language - Langage de définition des données) et DML (Data Manipulation Language - Langage de manipulation des données) pour HSQLDB, vous devez consulter la syntaxe supportée et modifier les déclarations (la grammaire) en conséquence.

Certains mots sont réservés par le standard et ne peuvent pas être utilisés comme des noms de tables ou de colonnes. Par exemple, le mot POSITION est une fonction définie par les standards d'un rôle similaire à String.indexOf() en langage Java, et est donc réservé. A l'heure actuelle, HSQLDB ne vous empêchera pas d'utiliser un mot réservé si il ne le gère pas ou ne le connait pas. Par exemple BEGIN est un mot réservé qui n'est pas aujourd'hui pris en charge par HSQLDB et est donc permis comme nom de table ou de colonne. Vous devez éviter l'usage de tels mots, comme les versions futures de HSQLDB seront amenées à les prendre en charge, rejetants alors vos définitions de tables et vos requêtes. La liste complête des mots réservés du SQL est dans la source de la classe `org.hsqldb.Token`

HSQLDB prend en charge également des mots-clés et expressions qui ne font pas partie du standard SQL. Des expressions telles que `SELECT TOP 5 FROM ..`, `SELECT LIMIT 0 10 FROM ...` ou `DROP TABLE mytable IF EXISTS` sont parmi quelques exemples.

Tous les mots-clés peuvent être utilisés pour les objets de base de données s'il sont encadrés par des guillemets doubles.

Contraintes et Index

Contraintes de clefs primaires

Avant la version 1.7.0, une instruction `CONSTRAINT <name> PRIMARY KEY` était traduite par le moteur en un index unique, et de plus, une colonne cachée était ajoutée à la table avec un autre index unique. Depuis la version 1.7.0 les deux contraintes de clef primaire simple et multicolonnes sont gérées. Elles sont autorisées par un index unique sur la ou les colonne(s) de clef primaire. Aucune autre colonne cachée n'est nécessaire pour ces index.

Contrainte UNIQUE

Conformément au standard SQL, une contrainte UNIQUE sur une colonne simple signifie que deux valeurs ne peuvent être égales à moins que l'une d'entre elles soit NULL. Ce qui signifie que vous pouvez avoir une ou plusieurs lignes ou la valeur de cette colonne est NULL.

Une contrainte UNIQUE sur des colonnes multiples (c1, c2, c3, ...) signifie qu'il ne peut pas y avoir deux jeux de valeurs identiques sauf si l'une des valeurs est NULL. Chaque colonne peut contenir des valeurs répétées. L'exemple suivant satisfait à une contrainte UNIQUE sur les deux colonnes:

Exemple 2.1. Valeurs de colonnes satisfaisant à une contrainte UNIQUE sur deux colonnes

1, 2
2, 1
2, 2
NULL, 1
NULL, 1
1, NULL
NULL, NULL
NULL, NULL

Depuis la version 1.7.2 le comportement des contraintes et index UNIQUE avec

le respect de la valeur NULL a changé pour se conformer aux standards SQL. Une ligne dans laquelle une des valeurs pour n'importe quelle colonne de la contrainte UNIQUE est NULL peut toujours être ajoutée à la table. Donc plusieurs lignes peuvent contenir les mêmes valeurs pour les colonnes de la contrainte UNIQUE si l'une des valeurs est NULL.

Index uniques

Dans la version 1.8.0, les index UNIQUE définis par l'utilisateur peuvent encore être déclarés mais ils sont obsolètes. Au lieu de l'index, vous devez utiliser une contrainte UNIQUE.

CONSTRAINT <Nom de l'index> UNIQUE créé toujours en interne un index unique sur les champs, comme dans les versions précédentes. Il a donc exactement le même effet que la déclaration index UNIQUE, maintenant obsolète.

Clés externes

Depuis la version 1.7.0, HSQLDB a implémenté les clés externes à colonne simple et multiples. Une clé externe peut aussi être spécifiée en référence à une table cible sans avoir à nommer la ou les colonnes cibles. Dans ce cas la ou les colonnes de clé primaire de la table cible sont utilisées comme les colonnes référencées. Les types des champs de chaque paire Colonne de référence / Colonne référencée doivent être identiques. Lors de la déclaration d'une clé externe, une contrainte unique (ou clé primaire) doit exister dans les colonnes référencées de la table contenant la clé primaire. Un index non unique est créé automatiquement sur les colonnes référencées. Par exemple:

```
CREATE TABLE enfant(c1 INTEGER, c2 VARCHAR, FOREIGN KEY (c1, c2) REFERENCES parent(p1, p2));
```

Il doit y avoir une contrainte UNIQUE sur les colonnes (p1, p2) dans la table nommée "parent". Un index non unique est automatiquement créé sur les colonnes (c1, c2) dans la table nommée "enfant". Les colonnes p1 et c1 doivent être du même type (INTEGER). Les colonnes p2 et c2 doivent également être du même type (VARCHAR).

Index et vitesse d'exécution des requêtes

HSQLDB ne fait pas usage d'index pour améliorer le tri dans l'affichage des requêtes. Mais les index jouent un rôle crucial dans l'amélioration de la vitesse d'exécution des requêtes. Si aucun index n'est utilisé dans une requête portant sur une seule table, comme une requête suppression (DELETE), alors toutes les lignes de la table devront être examinées. Avec un index sur un des champ figurant dans la clause WHERE, il est souvent possible de partir directement de la première ligne concernée, et ainsi réduire le nombre de lignes examinées.

Les index sont encore plus importants dans le cas de jointures entre plusieurs

tables. `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2` est exécutée en prenant les lignes de t1 une par une et en trouvant une ligne correspondante dans t2. S'il n'y a pas d'index dans t2.c2, alors pour chaque ligne de t1, toutes les lignes de t2 devront être vérifiées. Alors qu'avec un index, une ligne correspondante peut être trouvée en une fraction de seconde. Et si la requête comporte une condition sur t1, i.e., `SELECT ... FROM t1 JOIN t2 ON t1.c1 = t2.c2 WHERE t1.c3 = 4` alors un index sur t1.c3 supprimera le besoin de vérifier toutes les lignes de t1 une à une, et réduira le temps d'exécution à moins d'une milliseconde par enregistrement. En effet, si t1 et t2 contiennent chacune 10 000 lignes, la requête sans index devra vérifier 100 000 000 de combinaisons de lignes. Avec un index sur t2.c2, cela se retrouve réduit à 10 000 vérifications de lignes et recherches dans l'index. Avec l'index additionnel sur t2.c2, en moyenne 4 lignes seulement sont vérifiées avant d'obtenir la première ligne de résultat.

Les index sont créés automatiquement pour les clés primaires et les champs sans doublons. Pour en définir d'autres utilisez la commande CREATE INDEX.

Notez que dans HSQLDB un index unique sur plusieurs champs peut être utilisé en interne comme un index avec doublons sur la première colonne de la liste. Par exemple: `CONSTRAINT nom1 UNIQUE (c1, c2, c3);` signifie qu'il y a l'équivalent de `CREATE INDEX nom2 ON unetable(c1);`. Aussi vous n'avez pas besoin de spécifier un autre index si vous avez besoin de celui du premier champ de la liste.

Dans la version 1.8.0, un index multicolonnes accélerera les requêtes qui contiennent des jointures ou des valeurs dans TOUTES les colonnes. Vous n'avez pas besoin de déclarer d'autres index individuels sur ces champs à moins que vous n'utilisiez des requêtes qui effectuent des recherches seulement sur une partie de ces champs. Par exemple, les lignes d'une table qui a une clé primaire ou une contrainte UNIQUE sur trois champs, ou même un simple index sur ces champs, peuvent être trouvées efficacement quand les valeurs pour ces trois champs sont spécifiées dans la clause WHERE. Par exemple, `SELECT ... FROM t1 WHERE t1.c1 = 4 AND t1.c2 = 6 AND t1.c3 = 8` utilisera un index sur `t1(c1,c2,c3)` s'il existe.

Suite aux améliorations des index multi-clés, l'ordre de déclaration des champs dans l'index ou la contrainte a moins d'effet qu'avant sur la vitesse des recherches. Cependant si le champ qui contient le plus de valeurs différentes est en première place, la recherche sera sensiblement plus rapide

Un index multichamps n'accélèrera pas seulement les requêtes sur le second ou le troisième champ. Le premier champ doit être spécifié dans les conditions JOIN .. ON ou WHERE.

La vitesse d'exécution des requêtes dépend beaucoup de l'ordre des tables dans les clauses JOIN .. ON ou FROM. Par exemple la deuxième requête ci-dessous sera plus rapide avec de grandes tables (par le fait qu'il y ait un index sur `TB.COL3`). La raison est que TB.COL3 peut être évaluée très rapidement si la recherche s'applique sur la première table (et s'il y a un index sur TB.COL3):

```
....(TB est une très grande table avec seulement quelques lignes ou TB.COL3 = 4)
```

```
   SELECT * FROM TA JOIN TB ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;
```

```
   SELECT * FROM TB JOIN TA ON TA.COL1 = TB.COL2 AND TB.COL3 = 4;
```

La règle générale est de mettre d'abord la table dont un des champs a des conditions restrictives.

La version 1.7.3 fournit des index automatiques à la volée pour les vues et sous-sélections utilisées dans une requête. Un index est ajouté à une vue quand elle est jointe à une table ou à une autre vue.

Condition Where ou jointure ?

Utiliser des conditions WHERE pour joindre des tables revient à réduire la vitesse d'exécution. Par exemple la requête suivante sera généralement lente, même indexée:

```
   SELECT ... FROM TA, TB, TC WHERE TC.COL3 = TA.COL1 AND TC.COL3=TB.COL2 AND TC.COL4 = 1
```

La requête implique que `TA.COL1 = TB.COL2` sans spécifier explicitement cette condition. Si TA et TB contiennent chacune 100 lignes, 10 000 lignes seront jointes à TC avant d'appliquer les conditions de colonnes, et ceci bien qu'il y ait des index sur les colonnes jointes. Avec le mot-clé JOIN, la condition `TA.COL1 = TB.COL2` doit être explicite. Elle restreindra la combinaison de lignes de TA et TB avant qu'elle ne soit jointe à TC, aboutissant plus rapidement sur de grandes tables.

```
   SELECT ... FROM TA JOIN TB ON TA.COL1 = TB.COL2 JOIN TC ON TB.COL2 = TC.COL3 WHERE TC.COL4 = 1
```

La requête pourrait être grandement accélérée si l'ordre des tables jointes était changé, pour que `TC.COL1 = 1` soit appliqué en premier et que moins de lignes soient jointes ensemble:

```
   SELECT ... FROM TC JOIN TB ON TC.COL3 = TB.COL2 JOIN TA ON TC.COL3 = TA.COL1 WHERE TC.COL4 = 1
```

Dans l'exemple précédent le moteur applique automatiquement `TC.COL4 = 1` à TC et joint seulement aux autres tables le jeu de lignes qui satisfait à cette condition. Les index sur `TC.COL4`, `TB.COL2` et `TA.COL1` seront utilisés s'ils existent et accélèreront encore la requête.

Exemple commenté :

- Requêtes SQL différence de deux tables

(http://user.services.openoffice.org/fr/forum/ftopic15809)

Sous-requêtes et Jointures

Utiliser les jointures et définir l'ordre des tables pour un maximum de performances s'applique à tous les domaines. Par exemple, la deuxième requête ci dessous doit être plus rapide s'il y a des index sur TA.COL1 et TB.COL3

Exemple 2.2. Comparaison de requêtes

```
SELECT ... FROM TA WHERE TA.COL1 = (SELECT MAX(TB.COL2) FROM TB WHERE TB.COL3 = 4)
```

```
SELECT ... FROM (SELECT MAX(TB.COL2) C1 FROM TB WHERE TB.COL3 = 4) T2 JOIN TA ON TA.COL1 = T2.C1
```

La deuxième requête transforme `MAX(TB.COL2)` en une table à une seule ligne avant de la joindre à TA. Avec un index sur TA.COL1, c'est très rapide. La première requête, quant à elle, testera chaque ligne et évaluera `MAX(TB.COL2)` encore et encore.

Types de champs et opérations arithmétiques

Tous les types de colonnes de table pris en charge par HSQLDB peuvent être indexés et possèdent des fonctionnalités comparables. Les types de champ peuvent être explicitement convertis en utilisant la fonction de librairie CONVERT(), mais la plupart du temps ils sont convertis automatiquement. Il est recommandé de ne pas utiliser d'index sur les types LONGVARBINARY, LONGVARCHAR et OTHER, puisque ces index ne seront sans doute plus implémentés dans les versions futures.

Les versions précédentes de HSQLDB ne prenaient que peu en charge les opérations arithmétiques. Par exemple, il n'était pas possible d'insérer `10/2,5` dans aucun champ DOUBLE ou DECIMAL. Depuis la version 1.7.0, toutes les opérations sont possibles avec les règles suivantes:

TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC et DECIMAL (sans point décimal) sont de type entier et correspondent avec byte, short, int, long et BigDecimal dans Java. Le type SQL détermine les valeurs minimum et maximum qui peuvent être supportées pour chaque type de champ. Par exemple la valeur range (amplitude) pour TINYINT permet de -128 à +127, bien que le type de champ utilisé dans Java pour le manipuler soit `java.lang.Integer`.

REAL, FLOAT, DOUBLE sont tous interpretés comme double dans Java.

DECIMAL et NUMERIC sont interprétés comme `java.math.BigDecimal` et peuvent

être constitués d'un très grand nombre de chiffres.

Les nombres entiers

TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC et DECIMAL (sans le point décimal) sont complètement interchangeables en interne, et sans étranglement des données. Le résultat des opérations est retourné dans un `ResultSet` JDBC dépendant du type des opérandes et dans un des types Java suivants: `Integer`, `Long` ou `BigDecimal`. On peut utiliser la méthode `ResultSet.getXXXX()`pour retrouver des valeurs, aussi longtemps que le résultat de ces dernières peut être affiché dans le type résultant. Ce type est basé de manière arbitraire sur la requête et non sur ses lignes de résultat. Ce type ne change pas quand cette même requête qui retournait une ligne en retourne plusieurs, suite à l'ajout de données dans les tables.

Si la déclaration SELECT fait référence à un seul champ ou une fonction, le type retourné est le type correspondant au champ ou au type retourné par la fonction. Par exemple:

```
CREATE TABLE t(a INTEGER, b BIGINT); SELECT MAX(a), MAX(b) FROM t;
```

Affiche un résultat ou le type de la première colonne est `java.lang.Integer` et celui de la seconde colonne est `java.lang.Long`. Toutefois,

```
SELECT MAX(a) + 1, MAX(b) + 1 FROM t;
```

Retourne des valeurs de type `java.lang.Long` et `BigDecimal`, en résultat à une évolution uniforme du type pour toutes les valeurs de retour.

Il n'y a pas de limite intégrée sur la taille des valeurs entières intermédiaires à l'intérieur des expressions. Vous devez donc déterminer le type de la colonne du `ResultSet` et choisir la méthode `getXXXX()` appropriée pour pouvoir la gérer. Alternativement, vous pouvez utiliser la méthode `getObject()` , fondre la réponse en `java.lang.Number` et utiliser l'une des méthodes `intValue()` ou `longValue()` sur le résultat.

Le résultat d'une expression doit "tenir" dans la colonne cible pour pouvoir être stocké dans le champ. Sinon vous obtiendrez un message d'erreur. Par exemple le résultat de `123456789012345678901 2 / 12345678901234567890` peut être stocké dans tous les types de champs d'entiers, y compris une colonne TINYINT. Effectivement, c'est une petite valeur.

Autres types numériques

Dans les déclarations SQL, les nombres dotés d'un point décimal sont traités comme de type DECIMAL sauf s'ils sont écrits avec un exposant. `0.2` est donc

considéré comme DECIMAL alors que `0.2E0` est traité comme DOUBLE.

A l'utilisation de `PreparedStatement.setDouble()` ou `setFloat()` , la valeur est automatiquement traitée comme DOUBLE.

Quand un REAL, FLOAT ou DOUBLE (tous synonymes) font partie d'une expression, le type du résultat est DOUBLE.

D'autre part, si aucune valeur DOUBLE n'existe, et qu'une valeur DECIMAL ou NUMERIC fait partie de l'expression, le type du résultat est DECIMAL. Le résultat peut être récupéré depuis un `ResultSet` dans le type requis aussi longtemps qu'il peut y être représenté. Cela signifie que les valeurs DECIMAL peuvent être converties en DOUBLE, à moins qu'elles ne soient en dehors de la plage comprise entre `Double.MIN_VALUE` et `Double.MAX_VALUE`. Comme pour les valeurs entières, le résultat d'une expression doit "tenir" dans la colonne cible pour pouvoir être stocké dans le champ. Sinon vous obtiendrez un message d'erreur.

La distinction entre DOUBLE et DECIMAL est importante en cas de division. Quand les variables sont de type DECIMAL, le résultat est une valeur munie d'une échelle (nombre de chiffres à droite du point décimal) égale à la plus grande des échelles des variables. Avec une variable de type DOUBLE, l'échelle reflètera le résultat actuel de l'opération. Par exemple, `10.0/8.0` (DECIMAL) égale `1.2` mais `10.0E0/8.0E0` (DOUBLE) égale `1.25`. En dehors des opérations de division, les valeurs de type DECIMAL représentent l'arithmétique exacte; Lors de la multiplication, l'échelle résultante est la somme des échelles des deux variables.

Les trois types de valeurs REAL, FLOAT et DOUBLE sont tous stockés dans la base de données comme des objets `java.lang.Double`. Les valeurs spéciales telles que NaN ou +-l'infini sont aussi stockées et prises en charge. Ces valeurs peuvent être fournies à la base de données via les méthodes JDBC PreparedStatement. Elles sont relues dans des objets ResultSet.

Les types Bit et Boolean

Depuis la version 1.7.2, le type BIT est juste un alias pour BOOLEAN. L'image primaire du champ BOOLEAN est `'true'` ou `'false'` ('vrai' ou 'faux'), que ce soit comme type BOOLEAN ou comme une chaîne de caractères lorsqu'on l'utilise depuis JDBC. Ce type de champ peut aussi être initialisé en utilisant des valeurs de n'importe quel type numérique. Dans ce cas 0 est interprété comme `false` et n'importe quelle autre valeur, telle 1, est traduite en `true`.

Depuis la version 1.7.3 le type BOOLEAN se conforme aux standards SQL et prend en charge l'état UNDEFINED (indéfini) en plus de TRUE ou FALSE. Les valeurs NULL sont traitées comme indéfinies. Cette amélioration affecte les requêtes contenant la déclaration NOT IN. Veuillez lire le fichier texte test, TestSelfNot.txt, pour des exemples de requêtes.

Stockage et manipulation des objets Java

Depuis la version 1.7.2 la prise en charge s'est améliorée et chaque objet JAVA sérialisable peut être ajouté directement dans une colonne de type OTHER en utilisant les variations des méthodes de `PreparedStatement.setObject()`.

A but de comparaison et en ce qui concerne les index, deux objets Java sont considérés comme égaux à moins que l'un d'entre eux ne soit NULL. Vous ne pouvez pas chercher un objet spécifique ou réaliser une jointure sur une colonne de type OTHER

Veuillez noter que HSQLDB n'est pas une base de données relationnelle d'objets. Les objets Java peuvent seulement être stockés en interne et aucune autre opération que l'assignation entre colonnes de type OTHER ou Tester les NULL. Des tests tels que `WHERE object1 = object2`, ou `WHERE object1 = ?` ne signifient pas ce que vous pourriez escompter, puisque tous les objets non nuls satisfont à un tel test. Mais `WHERE object1 IS NOT NULL` est parfaitement acceptable.

Le moteur ne renvoie pas d'erreurs quand des valeurs de colonnes normales sont assignées aux colonnes d'objets Java (par exemple l'assignation d'une valeur INTEGER ou STRING à un tel champ par une déclaration SQL comme `UPDATE mytable SET objectcol = intcol WHERE ...`) mais c'est la rendre indisponible dans le futur. Aussi utilisez les colonnes de type OTHER seulement pour stocker vos objets, et pour rien d'autre.

Caractéristiques de taille, de précision et d'échelle

Avant la version 1.7.2, toutes les définitions de champ de table dotées d'une taille de colonne, d'un qualificatif précision ou échelle étaient acceptées et ignorées.

Depuis la version 1.8.0, de tels qualificatifs doivent se conformer aux standards SQL. Par exemple INTEGER(8) n'est maintenant plus accepté. Les qualificatifs sont toujours ignorés sauf si vous ajoutez une propriété à votre base de données. `SET PROPERTY "sql.enforce_strict_size" TRUE` imposera les tailles pour les types de champs CHARACTER ou VARCHAR et marquera chaque chaîne de caractères lors de l'insertion ou la mise à jour d'une colonne CHARACTER. Les qualificatifs Précision et échelle sont également imposés pour les types DECIMAL et NUMERIC. TIMESTAMP peut être seulement utilisé avec une précision de 0 ou 6.

Une proposition : Le transtypage d'une valeur à un type CHARACTER qualifié entrainera la troncature ou le remplissage que vous attendez. Ainsi un test tel que as `CAST (mycol AS VARCHAR(2)) = 'xy'` trouvera les valeurs commençant par 'xy'. C'est l'équivalent de `SUBSTRING(mycol FROM 1 FOR 2)` = 'xy'.

Casting a value to a qualified CHARACTER type will result in truncation or padding as you would expect. So a test such as `CAST (mycol AS VARCHAR(2)) = 'xy'`

will find the values beginning with 'xy'. This is the equivalent of `SUBSTRING(mycol FROM 1 FOR 2) = 'xy'`.

Séquences et Identity

Le mot clé SEQUENCE est apparu dans la version 1.7.2 avec un sous-ensemble de la syntaxe standard SQL 200n. La syntaxe SQL 200n correspondante pour les colonnes IDENTITY a aussi été ajoutée.

Champs Identifiants auto-incrémentés

Chaque table peut contenir une colonne auto-incrémentée, connue comme la colonne IDENTITY. Une colonne IDENTITY est toujours considérée comme la clé primaire de la table (et donc, les clés primaires multi-colonnes sont impossibles si une colonne IDENTITY existe). La prise en charge de `CREATE TABLE <NomDeLaTable>(<NomDeLaColonne> IDENTITY, ...)` a été ajoutée comme raccourci.

Depuis la version 1.7.2, la syntaxe standard SQL est utilisée par défaut, ce qui permet de préciser les valeurs initiales. La forme supportée est `<NomDeLaColonne> INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH n, [INCREMENT BY m])PRIMARY KEY, ...)`. La prise en charge des colonnes d'identités pour BIGINT a également été ajoutée. En conséquence, une colonne IDENTITY est seulement une colonne INTEGER ou BIGINT avec une valeur par défaut créée par un générateur séquentiel.

Quand vous ajoutez une nouvelle ligne à une telle table en utilisant la déclaration `INSERT INTO <NomDeLaTable> ...;` , vous pouvez utiliser la valeur NULL pour la colonne IDENTITY, ce qui aura pour effet une valeur auto-générée pour ce champ. La fonction `IDENTITY()` renvoie la dernière valeur ajoutée dans n'importe quelle colonne IDENTITY par cette connexion. Utilisez la déclaration SQL `CALL IDENTITY();` pour récupérer cette valeur. Si vous voulez utiliser la valeur pour un champ d'une table enfant, vous pouvez utiliser `INSERT INTO <TableEnfant> VALEURS (...,IDENTITY(),...);`. Les deux types d'appel à `IDENTITY()` doivent être réalisés avant toute autre mise à jour ou des déclarations d'insertions resteront irrésolues dans la base de données.

La prochaine valeur IDENTITY à insérer peut être réglée par la déclaration

```
ALTER TABLE ALTER COLUMN <NomDeLaColonne> RESTART WITH <NouvelleValeur>;
```

Séquences

La syntaxe et l'utilisation de SQL 200n est différente de celle de nombreux moteurs de base de données. Les séquences sont créées par la commande `CREATE SEQUENCE` et leur valeur courante peut être modifiée à tout moment avec `ALTER SEQUENCE`. La valeur suivante d'une séquence est récupérée avec

l'expression `NEXT VALUE FOR <nomDuChamp>`. Cette expression peut être utilisée pour ajouter et mettre à jour des lignes de la table. Vous pouvez aussi l'utiliser dans les déclarations SELECT. Par exemple, si vous voulez énumérer les lignes d'une commande SELECT en ordre séquentiel, vous pouvez utiliser:

Example 2.3. Énumérer les lignes d'une commande SELECT en ordre séquentiel

```
SELECT NEXT VALUE FOR MaSequence, col1, col2 FROM MaTable WHERE ...
```

Veuillez noter que la sémantique des séquences est légèrement différente de celle définie par SQL 200n. Par exemple si vous utilisez la même séquence deux fois dans la même requête insertion, vous obtiendrez deux valeurs différentes, et non la même comme le requiert le standard.

Vous pouvez rechercher dans la table SYSTEM_SEQUENCES la prochaine valeur qui sera retournée pour chaque séquence définie. La colonne SEQUENCE_NAME contient le nom et le champ NEXT_VALUE contient la prochaine valeur à retourner.

Problèmes avec les transactions

HSQLDB prend en charge les transactions au niveau READ_UNCOMMITTED (Lecture seule non validée), également connu comme le niveau 0 Isolation de la transaction. Ceci signifie que durant la durée de vie de la transaction, les autres connexions pourront voir les changements appliqués aux données. La prise en charge des transactions fonctionne généralement correctement. Les rapports de bugs concernant les transactions en cours de validation lorsque la base se ferme brutalement ont été corrigés. Toutefois, les problèmes suivants pourraient être rencontrés dans le cas de connexions multiples à une base de données utilisant les transactions:

Si deux transactions modifient la même ligne, aucune exception n'est soulignée lorsque les deux transactions sont validées. Ceci peut être évité en construisant votre base de données de telle façon que la consistance des données de l'application ne dépende pas de la modification exclusive des données par une seule transaction. Vous pouvez définir une propriété de la base de données pour provoquer l'exception quand cela se produit.

```
SET PROPERTY "sql.tx_no_multi_rewrite" TRUE
```

Quand une commande `ALTER TABLE .. INSERT COLUMN` or `DROP COLUMN` aboutit à un changement de la structure de la table, la session en cours est validée. Si une transaction non validée engagée par une autre connexion a changé les données dans la table, il ne sera plus possible d'annuler les changements après la commande `ALTER TABLE` . Cela s'applique aussi aux commandes `ADD INDEX` ou `ADD CONSTRAINT` . Il est recommandé de n'utiliser ces commandes ALTER qu'une fois

assuré qu'aucune autre connexion n'effectue de transactions.

Lorsqu'une commande CHECKPOINT est émise, les transactions non validées peuvent être continuées, validées ou annulées. Toutefois, si la base de données n'est pas fermée proprement par la commande SHUTDOWN, toute transaction non validée par un COMMIT avant l'arrêt, retrouvera au démarrage suivant son état au moment du CHECKPOINT. Il est préconisé d'utiliser la commande CHECKPOINT soit quand aucun traitement en cours ne nécessite la validation de la transaction, soit quand il est accepté qu'un arrêt impromptu de la base n'affecte ses données.

Nouvelles fonctionnalités et changements

Dans les dernières versions jusqu'à la 1.8.0 beaucoup d'améliorations ont vu le jour pour une meilleure prise en charge de SQL. Elles sont listées au chapitre Syntaxe SQL , dans changelog_1_8_0 et changelog_1_7_2. Vous trouverez des fonctions et expressions comme POSITION(), SUBSTRING(), NULLIF(), COALESCE(), CASE ... WHEN .. ELSE, ANY, ALL etc. parmi eux. D'autres améliorations ne sont peut être pas évidentes dans la documentation mais il y a pu avoir des changements de comportement depuis les versions précédentes. Les plus importantes parmi elles sont la manipulation de NULL dans les jointures (les colonnes nulles ne sont plus jointes) et les jointures externes (les résultats sont maintenant corrects). Vous devez tester vos applications avec la dernière version pour vous assurer qu'elles ne s'en remettent pas à un comportement incorrect d'un moteur obsolète. Le moteur continuera d'évoluer vers la prise en charge complète standard SQL dans les futures versions, aussi il est préférable de n'utiliser aucune fonctionnalité non standard de la présente version.

Exemples commentés

Exemples commentés :

- Condition Where ou jointure ? : Requêtes SQL différence de deux tables (http://user.services.openoffice.org/fr/forum/ftopic15809)