Litebase

The Litebase Companion

Version 2.00

Table of Contents

PART I – LITEBASE....5
LITEBASE....6
OVERVIEW....6
NEW TABLE FORMAT....7
LITEBASE MULTIPLE LANGUAGES....10
LIMITATIONS AND USAGE....10
Memory Card Support....13
Using Threads....14
Logging and Debugging....14
Compatibility....15
LITEBASE SQL FUNCTIONS....16
LITEBASE RESERVED WORDS....19
LITEBASECONNECTION CLASS....29
GetInstance()....29
GetInstance()....29
getInstance()....29
getSourcePath()....30
execute()....30
executeUpdate()....33
executeQuery()....38
prepareStatement()....43
getCurrentRowId()....43
getRowCount()....44
setRowInc()....44
convert()....45
exists()....45
closeAll()....45
purge()....46
getRowCountDeleted()....46
getRowIterator()....46
getLogger()....47
setLogger()....47
getDefaultLogger()....47
DeleteLogFiles()....48
processLogs()....48
recoverTable()....48
getSlot()....49
isOpen()....49
dropDatabase()....49
RESULTSET CLASS....50
getResultSetMetaData()....50
close()....50
beforeFirst()....50
afterLast()....50
first()....50
last()....51
next()....51
prev()....51

getShort()....51
getShort()....51
getInt()....51
getInt()....51
getLong()....52
getLong()....52
getFloat()....52
getFloat()....52
getDouble()....52
getDouble()....52
getChars()....52
getChars()....53
getString()....53
getString()....53
getStrings()....53
getStrings()....54
getDate()....58
getDate()....58
getDateTime()....58
getDateTime()....59
absolute()....59
relative()....59
getRow()....59
setDecimalPlaces()....59
getRowCount()....59
isNull()....60
isNull()....60
PREPAREDSTATEMENT CLASS....61
executeQuery()....61
executeUpdate()....61
setShort()....61
setInt()....62
setLong()....62
setFloat()....62
setDouble()....62
setString()....62
setBlob()....63
setDate()....63
setDateTime()....63
setDateTime()....64
setNull()....64
clearParameters()....65
toString()....65
RESULTSETMETADATA CLASS....66
getColumnCount()....66
getColumnDisplaySize()....66
getColumnLabel()....67
getColumnType()....67
getColumnTypeName()....68
getColumnTableName()....68

getColumnTableName()....68
hasDefaultValue()....68
hasDefaultValue()....68
isNotNull()....69
isNotNull()....69
ROWITERATOR CLASS....69
data....70
rowid....70
attr....70
rowNumber....70
next()....70
nextNotSynced()....70
setSynced()....70
close()....70
reset()....71
getShort()....71
getInt()....71
getFloat()....72
getDouble()....72
getString()....72
getDate()....72
getDateTime()....73
getBlob()....73
isNull()....73
MIGRATION FROM DIFFERENT TABLE FORMATS....74
MIGRATING THE OLD TABLES....74
LITEBASE CONDUIT....76

....79
PART II – APPENDIXES 79
APPENDIX I – COPYRIGHT....80
APPENDIX II – NEWS....81

PART I – LITEBASE

LITEBASE

OVERVIEW

Litebase is a full-featured database for TotalCross. As most databases, it uses the SQL language to let you manipulate data.

The normal way to access files in TotalCross is through the `File`, `PDBFile`, `ResizeStream` and `DataStream` classes. When using Litebase, you will have to forget these classes and use only the ones provided by the `litebase.*` packages. Trying to use the one or more of these classes in conjunction with the Litebase ones within the same file can lead to data corruption and is not recommended. Moreover, some platforms do not let the same file be used concurrently.

You can run Litebase in desktop, either as an applet, a Linux application, as a windows 2000 (and up) application, or in all the supported TotalCross platforms. So, its easy to create table files in the desktop that can be later synchronized with the device. The files created are interchangeable on all supported platforms.

The packages that belong to this library are specified below:

- `litebase`: contains the database classes that can be used to manipulate the plain files via SQL
- `litebase.ui`: contains some useful user interface classes that can be used with the driver
  - `DBListBox`: a class used to show a single `listbox` of a `ResultSet` that returns multiple columns. Because it extends `totalcross.ui.ListBox`, it can be placed in a `ComboBox` too.

Both packages are inside the Litebase files.

These samples are provided under `samples`:

- `addressbook`: a simple address book application, which demonstrates the use of the `Grid`.
- `bench`: checks the search speed of Litebase. Note that it can take some minutes to run on a device.
- `testcases`: `TestUnit` testcases for Litebase: it will give you an overall idea of what is supported. This can take several minutes to run on a device!
- `sqlconsole`: a console for issuing SQL commands. In this application, it is impossible to use prepared statements and blobs. Using this program you can view the results of queries and create, drop and change tables.
- `photodb`: a small sample application showing how to use blobs to store and recover pictures in a database.

- `conduit`: a conduit application to synchronize tables between the desktop and palm devices.
- `migration`: a program used to aid the user in the migration from tables whose format is compatible from Litebase 2.11 to Litebase 2.14 to the format used from Litebase 2.20 to Litebase 2.26.

Next we provide a sample program that uses Litebase:

```
// Create the driver's instance.
LitebaseConnection driver = LitebaseConnection.getInstance("Test");

// Create the table and the index.
try
{
   driver.execute("CREATE TABLE person (name char(30), sal double, age int)");
   driver.execute("CREATE INDEX idx_name ON person(name)");
} catch (AlreadyCreatedException exception) {}

// Insert some Values.
driver.executeUpdate("INSERT INTO person VALUES ('Michelle', 3000.0, 31)");
driver.executeUpdate("INSERT INTO person VALUES ('Simone', 3000.0, 33)");

// Do a query.
ResultSet rs = driver.executeQuery("SELECT * FROM person WHERE name = 
                                                                    'Michelle'");
if (rs.next())
{
   String name = rs.getString("name");
   double sal = rs.getDouble(2);
   ...
}
rs.close();
```

NEW TABLE FORMAT

Background: new Palm OS devices have a non-volatile memory (in other words, a pen-drive inside of it): Treo 650, Zire 22, Tungsten E2, Tungsten T|X. Many users complained about data corruption, slowness, and resets in these devices. After some research, a solution that fixed all these problems was found: the write of data **directly** to the non-volatile memory (AKA flash-memory). Given that the flash memory supports any kind of file, not only pdb nor prc files, it was decided to change the format of the tables and

indices: instead of using pdb files, in Litebase 1.00 new formats of files were created for it: db (database), dbh (database header), idk (index keys) and idr (index repetitions). These are like plain files, in contrast to PDB files, which have a predefined structure. In Litebase 2.00, the string fields of the database are now stored in a separate file .dbo (database objects). This is necessary to save space from the records, since strings may use much less space than the one declared when creating a table and all the records have fixed-size length. A similar approach is used for blob fields: they are stored in the same file. Moreover, .the header is now inside .db, that is, .dbh does not exist anymore. Another file that may not exist is .idr, if the index does not have repeated keys.

The advantages of the new model: all resets and data corruption on NVFS Palm devices are definitively gone, and the speed of Litebase on the other devices is much higher.

The drawbacks (for Palm OS devices only): table creation and some queries are slower because now Litebase is dealing with flash-memory (it is obvious that a pen drive is slower than a desktop computer main memory); secondly, it's now IMPOSSIBLE to synchronize data using hotsync since hotsync does not support the installation of anything besides pdb/prc (this is a stupid limitation). To bypass this problem, there is a conduit that will install the files in the flash-memory on device, and also get the data from it, just like hotsync. The tablecopier/LitebaseConduit uses the Conduit API to perform this task. Please read the chapter that describes this program.

Given that the database format from previous Litebase versions has changed, it was decided to create a migration method for customers that have the application deployed. There's also a chapter in this book that explains what a TotalCross developer has to do.

The index file format was also changed in Litebase 1.00. Instead of a ordered multi-part vector, a specialized B-Tree was implemented. Due to this, the whole table's column were stored in the tree keys (in other words, in the file); so, a big increase in the index size of columns of the CHAR type happened.

On version 1.1, some improvements were implemented on index file creation. The index file size now is less than the old one. Whenever the index format is changed, to make use of these and use the correct format, **it necessary to recreate all the indices**. One way to do this is just drop the index manually that the Litebase will recover them automatically. So, a simple code fragment is provided below that shows how to drop all the indices of an application whose tables are stored in `/Litebase_DBs/` and application id is `ACTv`:

```
// ......
import totalcross.sys.*;
import totalcross.io.*;

// ......
try
{
   // The folder where the tables files are stored.
   String sourcePath = "/Litebase_DBs/",
```

```
   // The application id of your database.
          appCreatorId = "ACTv";

   // Lists all the files of the given folder.
   String[] files = new File(sourcePath, File.READ_WRITE, 1).listFiles();

   Vm.debug("source path: " + sourcePath);

   if (files != null)
   {
      int size = files.length;
      String name;

      Vm.debug("size: " + size);
      while (--size >= 0)
      {
         name = files[size];

         // Files of this application.
         if (name.startsWith(appCreatorId + '-' )
          && (name.endsWith(".idk") || name.endsWith(".idr")))
         {
            new File(sourcePath + name).delete();
            Vm.debug("Index file deleted: " + name);
         }
      }
   }
}
catch (IOException exception)
{
   // ...
}
```

On version 2.00 and above, the contents of CHAR and VARCHAR (implemented in Litebase 2.00) are not stored in the indices anymore. This way, the indices files are much smaller. Instead, the position of the CHAR or VARCHAR fields in the .dbo file is stored in the index.

LITEBASE MULTIPLE LANGUAGES

Starting on version 2.00, Litebase supports more than one language. Up to now, the supported languages are: English (the default language) and Portuguese. This feature puts Litebase's messages in your chosen language. To chose a language, it is just necessary to set the language field:

```
// Setting the language to Portuguese.
LitebaseConnection.language = LitebaseConnection.LANGUAGE_PT;
```

The possible parameters are:

`LitebaseConnection.LANGUAGE_EN:` English (Default)

`LitebaseConnection.LANGUAGE_PT:` Portuguese

To get the current language, it is only necessary to access the field language::

```
int language = LitebaseConnection.language;
```

LIMITATIONS AND USAGE

- The `LitebaseConnection` class is used to issue the SQL command for a specific set of tables, and is constructed using a creator id. All tables created by this instance will have the given creator id.
- If Litebase is to be used in an applet, the jars used some be signed in order to be used in a browser. Otherwise, it won't be possible to create or access tables.
- An index is created using the table name and appending a `$<column number>` to indices of one attribute or `&<internal code>` to indices of two or more attributes (Composed Index). Thus, **the table name is limited to 23 characters.** It is possible to create up to 32 composed indices.
- Indices are used during the evaluation of the **where** clause when:
    - The following operators are used:
        - and;
        - or;
        - ( );
        - not (are removed internally), and
        - like 'a%' (starts with);
    - The following operators are not used:
        - like '%a' (ends with), and
        - like '%a%' (index of).

- Indices are not used for all the where clause if it uses different boolean operators (nots are not counted as they are removed).
- If your where clause is of the form `A and B and C ... and Z`, or `A or B or C ... or Z`, to ensure that the indices will be applied to all columns that have indices, the columns using indices should be preferable be in the rightmost part of the clause.
- Litebase does not have the reserved word IN. A big OR with all the values in the set must be used instead.
- The composed indices are used only when there's an AND operation of the correspondent fields of the composed index in its creation order, and the fields use equals operator on comparison.
- Indices can also be used to compute **max** and **min** aggregation functions. In order to use them, the column must use have an index or be the first column of a composed index and the query cannot have group by. Moreover the query can't have a where clause or its where clause must be all solved using indices.
- Indices may also be used when using order by or group by clauses. In order to use them, the query also can't have a where clause or its where clause must be all solved using indices. Furthermore, the query can't have aggregations. Additionally, if there is only one column to be sorted, it must be a primary key column, the first column of a composed primary key column or the first column of an index if it is declare as not null. If there are more than one column to be sorted, they must be the fist part of the composed primary key or a composed index if they are declared as not null. Notice that columns that are not part of a primary key must be declared as not null since Litebase indices do not store nulls. Finally, the order of all columns to be ordered must be the same, either all of them are sorted in the ascending or in the descending order.
- Indices can be created after a table already contains elements. Note that the creation of the index can be a slow operation; it can take several minutes on big tables. The best method is to create all indices when there's no data in the table. However, If you plan to insert lots of data (above 20% of current size, or above 1000 rows), drop all indices, insert the data, then create them again.
- SQL commands are case insensitive.
- Primary Keys and composed primary keys are supported.
- Aggregation functions (**max, min, avg, sum, count**), **order by** and **group by** are supported.
  - SUM and AVG aggregate functions are not used with DATE, DATETIME, CHAR, CHAR NOCASE, VARCHAR, and VARCHAR NOCASE type fields.
  - COUNT can only be used with * as a parameter.
- The join operation is supported. Inner, Left and Right join are not supported. The only way to do a join is as the following sample:

  ```
  "Select * from table1, table2, tableN ..."
  ```

  The performance of a join can vary greatly depending on the order of the tables and the clauses in the where clause. So, if a join operation is taking too long, one should try changing these orders and see if the performance improves.

  Sometimes, the join speed can be improved if the clauses in the where clause use

simple indices. Moreover, if there is a restricted clause, this should be the last clause of the where clause. Additionally, the order of the clauses must be in such a way that the first clauses access the smallest tables. Last but not least, in a clause, the first field being accessed should be of the smallest table. These recommendations are some strategies that usually work. However, some joins usually perform better doing the reverse. Consequently, if the join is still very slow, one should try reversing the tables order and where clause order until finding a better result.

If the join is always very slow, it is recommended to do simple joins of two tables only filtering using the matching field (… `where table1.x = table2.x`) and then join the results and filters it “by hand”. Join will be highly improved in future Litebase versions.

- Null and Default values are supported.
- BLOB type is supported. Notice that blobs must be inserted in a table only via prepared statement, as a byte array. It also can't be used in functions, aggregations, where, having, order by, and group by clauses because it does not make sense to compare blobs. Therefore, they can't also be indexed. Moreover, they can't be displayed as a string by Litebase. Whenever you try to recover a BLOB field from a table as a string, null is returned instead.
- VARCHAR type is supported. Internally, CHAR and VARCHAR are treated in the same way in order to not let the table files become to big. If your application has memory problems, try to reduce SQL strings length changing field declarations from VARCHAR to CHAR.
- It is possible to use ' (single quote) inside strings. Just use \'.
  - Using prepared statement in batch operations is **3 to 4** times faster than using direct calls to LitebaseConnection.executeUpdate.
  - A query with no where clause, using all fields (*) or the table fields in the correct order beginning with the rowid is much faster than other queries that return all the rows. This happens because table data do not need to be searched. Since version 2.3, no temporary tables are created if the query do not have joins, aggregations or sortings, which improves performance and memory usage.
  - The table's file name is prefixed with the **creator id**. This is important to let two different programs use the same table name. Prefixing the table name with the creator id ensures that the files won't be overwritten (unless they match, which will occur if they don't obey the rules for creating and defining creator ids).
  - It is possible to use multiple connections. That is, it is possible to access two databases at the same time in different directories. When using Palm OS, Windows Mobile, BlackBerry and Android, it is possible to access tables in the flash memory and in the memory card at the same time.
  - `LitebaseConnection.exists()` does not load the table and indices. It only tests if the .db file exists in the current connection path. This prevents the program to abort if the table and/or indices are corrupted or are using an older format. It also works properly with already loaded tables.
  - Drop table also does not load the table files. It only searches for the table files and erase them. This also prevents the program to abort if the table and/or indices are corrupted or are using the old format. Moreover, it works properly with already loaded

tables.

- If Unicode characters are not to be used in the tables, it is possible to create them in order to use only ASCII characters in string types. This saves memory, disk space and makes the reading and writing operations faster. It is very important to say that a .dbo that has only strings has its size almost halved if it is stored in the ASCII mode. Notice that all the tables of a given connection have their strings stored in either ASCII or Unicode format, and it is not possible to open a table with a connection using a different format of string saving nor changing its format after creation.
- Relative paths can't be used with Litebase. That is, a relative path can't be passed to `Settings.dataPath` (remember that `Settings.appPath` can't be changed!) or as a new connection parameter. If this is done, an exception will be thrown.
- CHAR(1) columns can be created, bu it is much more efficient to use a short column with the char ASCII code.
- The tables should not be opened at the same time by two different connections. On Java SE, BlackBerry, Windows 32, Windows Mobile, and Palm, a file cannot be opened twice at the same time. So, an exception will be thrown if a table is opened at the same time by two different connections. However, on Android, Linux, and iPhone it is possible to open twice at the same time. Therefore, if a table is opened at the same time by two different connections, your application may behave unexpectedly.
- There are some limits that must be respected:
    - It is not possible to create a table with more than 128 columns, nor using more than 128 fields or columns in a clause;
    - A blob field can't be declared to be larger than 10 Mb;
    - It is not possible to create a table name with more than 23 characters. This limitation is due to the compatibility with palm OS;
    - A table can't have more than 32 composed indices;
    - The .dbo file can't be greater than 2 Gb. This will be changed as the PDAs are improved, and
    - A table with an index can't have more than ~1,3 million records with different index keys. Again, This will be changed as the PDAs are improved.

- **It is very important that you take a look at the test cases.**

Memory Card Support

It is possible to store databases on Palm OS memory cards. In the previous version of Litebase, to do this, you must specify “oncard” (case sensitive!) in `LitebaseConnection.getInstance()` method. The tables and indices will be stored/loaded from /LitebaseDBs folder.

It is NOT a good idea to create the tables on the card using the PDA: table writes are usually 5 times slower than in main memory, and you will notice that it is

UNFEASIBLE to do that with even a few records. The idea is to create the databases using a desktop computer with a card reader.

Note that, in the past, after you created an instance that is “oncard”, **all future instances would also be “oncard”**, even if you didn't specify it in the `getInstance()` method.

Now, it is also possible to use the memory card by giving its volume number in the data path passed to a new connection. Moreover, one can pass -1 as the volume number to use the last volume. From now on, one can use the memory card and the flash memory at the same time by using different connections. To see how to store tables on it, take a look at the `TestPalmMemoryCard` test case.

It must be noticed that on BlackBerry, writing operations using the memory card are faster than using the main memory. So, when using this device, it is highly recommended to create and populate the tables on the card and then copy the files to the main memory.

On BlackBerry, Android, and Windows Mobile, there is no slot. It is only necessary to pass the card path to the new connection or to the data path.

Finally, when using Palm devices, if slot 0 (zero) is used, the device will reset because it is not possible to store tables in the internal memory using Litebase 2.

Examples for Palm OS:

```
// Litebase 1: Use LitebaseConnection.getInstance(<creator id>, “oncard”).
LitebaseConnection driver = LitebaseConnection.getInstance(“Test”, “oncard”);


// Litebase 2: Use LitebaseConnection.getInstance(<creator id>, path).
LitebaseConnection driver = LitebaseConnection.getInstance(“Test”, "-1:\\data");
```

Using Threads

It is possible to use Litebase within threads. An user can access different connections in different threads. However, it is highly not recommended to access the same tables and connections within different threads. Doing this will probably crash your application. This was implemented this way to improve execution performance.

Logging and Debugging

Litebase supports logging. It will write to a file all the operations, such as queries, updates, deletes, inserts, and other methods of the LitebaseConnection class that you call. It can help Litebase developers a lot in finding the causes of a bug in the driver, but it must be used with care, because it will slow down Litebase and also take much of the device's memory (E.G.: a log file of the AllTests program takes more than 380kb). So, if you find a problem when using Litebase, follow these steps:

1. Add this line to your application before using Litebase:

```
LitebaseConnection.logger = LitebaseConnection.getDefaultLogger();
```

if your application does not use threads. If it uses threads, use

```
LitebaseConnection.setLogger(LitebaseConnection.getDefaultLogger());
```

2. Run your application until the error is found.
3. Synchronize the files to the desktop and find the file(s) named as Litebase_YYYYMMDDHHMMSS.CRTR.LOGS, where CRTR is the creator id of your application.
4. Zip the file(s) and open a support request at our CRMDesk system explaining what you have been doing, the error that occurred, which platform and device you were using, along with uploading the zipped log file and table files in a state after and before the execution of the log commands.
5. Don't forget to turn off the log, by commenting out the line above, unless you want to fill up all your PDA's space. To dispose you log, use `Logger.dispose(true)`. It is also necessary to set the Litebase logger to `null` doing:

```
LitebaseConnection.logger = null;
```

if your application does not use threads. If it uses threads, use

```
LitebaseConnection.setLogger(null);
```

Log files are never deleted, you must do it by yourself, calling `LitebaseConnection.deleteLogFiles()` for log files created with `LitebaseConnection.getDefaultLogger()`. They are no longer pdb files as it were in versions before 2.28.

6. The logger can also be something different from a normal text file. Please check `totalCross.util.Logger` for more information.
7. If you prefer using logging all the time, your PDA WILL run out of storage space. This may cause a Litebase operation failure and inconsistency between files and tables if you don't stop using Litebase for a while and erase log files before running out of space.

Compatibility

Litebase needs TotalCross to run on the devices. Moreover, one specific Litebase version does not run on all TotalCross versions. So, it is not recommended to run a Litebase version on incompatible TotalCross version. This can cause strange errors or even an application crash. The table below shows the ensured compatibility between TotalCross and Litebase versions.

TotalCross	Litebase
1.20	2.20
1.21	2.21
1.22	2.22

1.23	2.23
1.24	2.24
1.25	2.25 2.26 2.26a
1.27	2.27
1.28	2.28
1.30	2.30

LITEBASE SQL FUNCTIONS

In this section, you find all sql data type functions accepted by Litebase.

1. SHORT, INT, LONG, FLOAT and DOUBLE functions:

Function	Return Type	Description	ResultSet method
ABS(*field*)	SHORT	Returns the absolute value of the given SHORT field.	getShort() or getString()
ABS(*field*)	INT	Returns the absolute value of the given INT field.	getInt() or getString()
ABS(*field*)	LONG	Returns the absolute value of the given LONG field.	getLong() or getString()
ABS(*field*)	FLOAT	Returns the absolute value of the given FLOAT field.	getFloat() or getString()
ABS(*field*)	DOUBLE	Returns the absolute value of the given DOUBLE field.	getDouble() or getString()

Examples

```
// First create and fill the table.
// Note: These inserts statements will be used by all examples about SQL
//                                   functions.
LitebaseConnection driver =  LitebaseConnection.getInstance();
driver.execute("create table person(name char(16), amount int, amount1 short,
     amount2 long, amount3 float, amount4 double, birth Date, years DateTime)");
driver.executeUpdate("insert into person values ('Renato Novais', -12, -1,
           -100, -1.2, -456.0, '  2007/5-3  ', ' 2007/11-2    12:08:01:234 ')");
driver.executeUpdate("insert into person values ('indira gomes',13, -8, -25,
                                 5.2, -154.0, '2006/7/8 ', '2006/08-21 0:08')");
driver.executeUpdate("insert into person values ('Lucas Novais', -20, -456, 48
                             -5.9, -954.2, '2008/4/6', ' 2008/06/06  13:45 ')");
driver.executeUpdate("insert into person values ('Zenes Lima', -15, -54, -5698
                                -8.3, -456.5, '2005/9/12 ', '2005/01-4 1:50')");
```

```
// Selects with functions.
rs = driver.executeQuery("Select abs(amount) as a0, abs(amount1) as a1,
         abs(amount2) as a2, abs(amount3) as a3, abs(amount4) as a4 from person
                                                       where abs(amount)>13");
rs.getRowCount(); // Returns 2.
rs.next();
rs.getInt(1);     // Returns 20.
rs.getShort(2);   // Returns 456.
rs.getLong(3);    // Returns 48.
rs.getFloat(4);   // Returns 5.9.
rs.getDouble(5);  // Returns 954.2.
rs.next();
rs.getInt(1);      // Returns 15.
rs.getShort(2);   // Returns 54.
rs.getLong(3);    // Returns 5698 .
rs.getFloat(4);   // Returns 8.3.
rs.getDouble(5);  // Returns 456.5.
rs.close();
```

2. CHAR, VARCHAR and CHAR_NOCASE functions:

Function	Return Type	Description	**`ResultSet`** **method**
`UPPER(`*`field`*`)`	`CHAR`	Converts all lowercase letters in a character string to uppercase.	`getChars()` or `getString()`
`LOWER(`*`field`*`)`	`CHAR`	Converts all uppercase letters in a character string to lowercase.	`getChars() or getString()`

Examples

```
// Selects with functions.
rs = driver.executeQuery("Select amount, abs(amount) as a1, name,
                        lower(name) as u1, upper(name) as u2  from person
                        where abs(amount)>12 and UPPER(name) > 'INDIRA GOMES'");
rs.getRowCount(); // Returns 2.
rs.next();
rs.getInt(1);     // Returns -20.
rs.getInt(2);     // Returns 20.
rs.getString(3);  // Returns 'Lucas Novais'.
rs.getString(4);  // Returns 'lucas novais'.
rs.getString(5);  // Returns 'LUCAS NOVAIS'.
rs.next();
rs.getInt(1);     // Returns -15.
rs.getInt(2);     // Returns 15.
rs.getString(3);  // Returns 'Zenes Lima'.
rs.getString(4);  // Returns 'zenes lima'.
rs.getString(5);  // Returns 'ZENES LIMA'.
rs.close();
```

3. DATE and DATETIME functions:

Function	Return Type	Description	`ResultSet` method
`YEAR(`*`field`*`)`	`SHORT`	Gets subfield equivalent to year. Field CAN be DATE or DATETIME.	`getShort()` or `getString()`
`MONTH(`*`field`*`)`	`SHORT`	Gets subfield equivalent to month. Field CAN be DATE or DATETIME.	`getShort()` or `getString()`
`DAY(`*`field`*`)`	`SHORT`	Gets subfield equivalent to day. Field CAN be DATE or DATETIME.	`getShort()` or `getString()`
`HOUR(`*`field`*`)`	`SHORT`	Gets subfield equivalent to hour. Field MUST be DATETIME.	`getShort()` or `getString()`
`MINUTE(`*`field`*`)`	`SHORT`	Gets subfield equivalent to minute. Field MUST be DATETIME.	`getShort()` or `getString()`
`SECOND(`*`field`*`)`	`SHORT`	Gets subfield equivalent to second. Field MUST be DATETIME.	`getShort()` or `getString()`
`MILLIS(`*`field`*`)`	`SHORT`	Gets subfield equivalent to millis. Field MUST be DATETIME.	`getShort()` or `getString()`

Examples

```
// Selects with functions.
ResultSet rs = driver.executeQuery("Select month(years) as mon1, years from
                                                                    person");
rs.getRowCount();             // Returns 4.
rs.next(); rs.getShort(1);    // Returns 11.
rs.next(); rs.getShort(1);    // Returns 8.
rs.next(); rs.getShort(1);    // Returns 6.
rs.next(); rs.getShort(1);    // Returns 1.
rs.close();

rs = driver.executeQuery("Select year(years) as y1, years from person where
                                                         day(years) >= 6");
rs.getRowCount();          // Returns 2.
rs.next(); rs.getShort(1); // Returns 2006.
rs.next(); rs.getShort(1); // Returns 2008.
rs.close();

rs = driver.executeQuery("Select hour(years) as h1, day(birth) as d1 from person
                                 where month(birth) != 7 and hour(years) != 0");
rs.getRowCount(); // Returns 3.
rs.next();
rs.getShort(1);   // Returns 12.
rs.getShort(2);   // Returns 3.
rs.next();
rs.getShort(1);   // Returns 13.
rs.getShort(2);   // Returns 6.
rs.next();
rs.getShort(1);   // Returns 1.
rs.getShort(2);   // Returns 12.
rs.close();

rs = driver.executeQuery("Select  millis(years) as mil1, minute(years) as sec1
                                      from person where birth > '2005/9-12'");
```

```
rs.getRowCount(); // Returns 3.
rs.next();
rs.getShort(1);   // Returns 234.
rs.getShort(2);   // Returns 8.
rs.next();
rs.getShort(1);   // Returns 0.
rs.getShort(2);   // Returns 8.
rs.next();
rs.getShort(1);   // Returns 0.
rs.getShort(2);   // Returns 45.
rs.close();

rs = driver.executeQuery("Select  year(birth) as y1, month(birth) as m1
                        day(birth) as d1 from person where year(birth) = 2005");
rs.getRowCount(); // Returns 1.
rs.next();
rs.getShort(1);   // Returns 2005.
rs.getShort(2);   // Returns 9.
rs.getShort(3);   // Returns 12.
rs.close();

rs = driver.executeQuery("Select  hour(years) as h1, minute(years) as m1
second(years) as d1 from person where hour(years) >= 12");
rs.getRowCount(); // Returns 2.
rs.next();
rs.getShort(1);   // Returns 12.
rs.getShort(2);   // Returns 8.
rs.getShort(3);   // Returns 1.
rs.next();
rs.getShort(1);   // Returns 13.
rs.getShort(2);   // Returns 45.
rs.getShort(3);   // Returns 0.
rs.close();

rs = driver.executeQuery("Select  millis(birth) as mil, years from person");
// Throws exception: "Incompatible data type for the function call: millis",
// because birth is date type and millis is a function applied only for DateTime
// type.

rs = driver.executeQuery("Select  year(birth) as y1, month(birth) as m1,
                       day(birth) as d1 from person where second(birth) = 234");
// Throws exception: "Incompatible data type for the function call: second".

rs = driver.executeQuery("Select  month(birth) from person ");
// Throws exception: "An alias is required for the aggregate function column.
// Error position: 25."
```

LITEBASE RESERVED WORDS

Litebase, starting from version 2.0, has reserved words. So, you can not use these words as identifiers.

In the table below you have the reserved words from SQL2003 standard and from Litebase. Many words from SQL2003 standard are not in Litebase reserved words list, but we strongly recommend that you do not use these words, because in a near future they may become a Litebase reserved word.

Although DISTINCT is a reserved word, it is not currently used by Litebase. Group by with no aggregated function should be used instead to simulate DISTINCT behavior.

SQL Words	Litebase	SQL2003
ABS	reserved	
ADD	reserved	reserved
ALL		reserved
ALLOCATE		reserved
ALTER	reserved	reserved
AND	reserved	reserved
ANY		reserved
ARE		reserved
ARRAY		reserved
AS	reserved	reserved
ASC	reserved	
ASENSITIVE		reserved
ASYMMETRIC		reserved
AT		reserved
ATOMIC		reserved
AUTHORIZATION		reserved
AVG	reserved	
BEGIN		reserved
BETWEEN		reserved
BIGINT		reserved
BINARY		reserved
BLOB	reserved	reserved
BOOLEAN		reserved
BOTH		reserved
BY	reserved	reserved
CALL		reserved
CALLED		reserved
CASCADED		reserved

SQL Words	Litebase	SQL2003
CASE		reserved
CAST		reserved
CHAR	reserved	reserved
CHARACTER		reserved
CHECK		reserved
CLOB		reserved
CLOSE		reserved
COLLATE		reserved
COLUMN		reserved
COMMIT		reserved
CONDITION		reserved
CONNECT		reserved
CONSTRAINT		reserved
CONTINUE		reserved
CORRESPONDING		reserved
COUNT	reserved	
CREATE	reserved	reserved
CROSS		reserved
CUBE		reserved
CURRENT		reserved
CURRENT_DATE		reserved
CURRENT_DEFAULT_TRANSFORM_GROUP		reserved
CURRENT_PATH		reserved
CURRENT_ROLE		reserved
CURRENT_TIME		reserved
CURRENT_TIMESTAMP		reserved
CURRENT_TRANSFORM_GROUP_FOR_TYPE		reserved
CURRENT_USER		reserved
CURSOR		reserved
CYCLE		reserved
DATE	reserved	reserved

SQL Words	Litebase	SQL2003
DATETIME	reserved	
DAY	reserved	reserved
DEALLOCATE		reserved
DEC		reserved
DECIMAL		reserved
DECLARE		reserved
DEFAULT	reserved	reserved
DELETE	reserved	reserved
DEREF		reserved
DESC	reserved	
DESCRIBE		reserved
DETERMINISTIC		reserved
DISCONNECT		reserved
DISTINCT	reserved	reserved
DO		reserved
DOMAIN		reserved
DOUBLE	reserved	reserved
DROP	reserved	reserved
DYNAMIC		reserved
EACH		reserved
ELEMENT		reserved
ELSE		reserved
ELSEIF		reserved
END		reserved
ESCAPE		reserved
EXCEPT		reserved
EXEC		reserved
EXECUTE		reserved
EXISTS		reserved

SQL Words	Litebase	SQL2003
EXIT		reserved
EXTERNAL		reserved
FALSE		reserved
FETCH		reserved
FILTER		reserved
FLOAT	reserved	reserved
FOR		reserved
FOREIGN		reserved
FREE		reserved
FROM	reserved	reserved
FULL		reserved
FUNCTION		reserved
GET		reserved
GLOBAL		reserved
GRANT		reserved
GROUP	reserved	reserved
GROUPING		reserved
HANDLER		reserved
HAVING	reserved	reserved
HOLD		reserved
HOUR	reserved	reserved
IDENTITY		reserved
IF		reserved
IMMEDIATE		reserved
IN		reserved
INDEX	reserved	
INDICATOR		reserved
INNER		reserved
INOUT		reserved
INPUT		reserved

SQL Words	Litebase	SQL2003
INSENSITIVE		reserved
INSERT	reserved	reserved
INT	reserved	reserved
INTEGER		reserved
INTERSECT		reserved
INTERVAL		reserved
INTO	reserved	reserved
IS	reserved	reserved
ITERATE		reserved
JOIN		reserved
KEY	reserved	reserved
LANGUAGE		reserved
LARGE		reserved
LATERAL		reserved
LEADING		reserved
LEAVE		reserved
LEFT		reserved
LIKE	reserved	reserved
LOCAL		reserved
LOCALTIME		reserved
LOCALTIMESTAMP		reserved
LONG	reserved	
LOOP		reserved
LOWER	reserved	
MATCH		reserved
MAX	reserved	
MEMBER		reserved
MERGE		reserved
METHOD		reserved
MILLIS	reserved	

SQL Words	Litebase	SQL2003
MIN	reserved	
MINUTE	reserved	reserved
MODIFIES		reserved
MODIFY		reserved
MODULE		reserved
MONTH	reserved	reserved
NATIONAL		reserved
NATURAL		reserved
NCHAR		reserved
NCLOB		reserved
NEW		reserved
NO		reserved
NOCASE	reserved	
NONE		reserved
NOT	reserved	reserved
NULL	reserved	reserved
NUMERIC		reserved
OF		reserved
OLD		reserved
ON	reserved	reserved
ONLY		reserved
OPEN		reserved
OR	reserved	reserved
ORDER	reserved	reserved
OUT		reserved
OUTER		reserved
OUTPUT		reserved
OVER		reserved
OVERLAPS		reserved
PARAMETER		reserved

SQL Words	Litebase	SQL2003
PARTITION		reserved
PRECISION		reserved
PREPARE		reserved
PRIMARY	reserved	reserved
PROCEDURE		reserved
RANGE		reserved
READS		reserved
REAL		reserved
RECURSIVE		reserved
REF		reserved
REFERENCES		reserved
REFERENCING		reserved
RELEASE		reserved
RENAME	reserved	
REPEAT		reserved
RESIGNAL		reserved
RESULT		reserved
RETURN		reserved
RETURNS		reserved
REVOKE		reserved
RIGHT		reserved
ROLLBACK		reserved
ROLLUP		reserved
ROW		reserved
ROWS		reserved
SAVEPOINT		reserved
SCOPE		reserved
SCROLL		reserved
SEARCH		reserved
SECOND	reserved	reserved
SELECT	reserved	reserved

SQL Words	**Litebase**	**SQL2003**
SENSITIVE		reserved
SESSION_USER		reserved
SET	reserved	reserved
SHORT	reserved	
SIGNAL		reserved
SIMILAR		reserved
SMALLINT		reserved
SOME		reserved
SUM	reserved	
SPECIFIC		reserved
SPECIFICTYPE		reserved
SQL		reserved
SQLEXCEPTION		reserved
SQLSTATE		reserved
SQLWARNING		reserved
START		reserved
STATIC		reserved
SUBMULTISET		reserved
SYMMETRIC		reserved
SYSTEM		reserved
SYSTEM_USER		reserved
TABLE	reserved	reserved
TABLESAMPLE		reserved
THEN		reserved
TIME		reserved
TIMESTAMP		reserved
TIMEZONE_HOUR		reserved
TIMEZONE_MINUTE		reserved
TO	reserved	reserved

SQL Words	Litebase	SQL2003
TRAILING		reserved
TRANSLATION		reserved
TREAT		reserved
TRIGGER		reserved
TRUE		reserved
UNDO		reserved
UNION		reserved
UNIQUE		reserved
UNKNOWN		reserved
UNNEST		reserved
UNTIL		reserved
UPDATE	reserved	reserved
UPPER	reserved	
USER		reserved
USING		reserved
VALUE		reserved
VALUES	reserved	reserved
VARCHAR		reserved
VARYING		reserved
WHEN		reserved
WHENEVER		reserved
WHERE	reserved	reserved
WHILE		reserved
WINDOW		reserved
WITH		reserved
WITHIN		reserved
WITHOUT		reserved
WORK		reserved
WRITE		reserved
YEAR	reserved	reserved

LITEBASECONNECTION CLASS

This class is the one used to issue SQL commands. It cannot be directly instantiated, only using the `getInstance()` methods.

If one tries to issue methods in an already closed connection, an `IllegalStateException` will be thrown. A `SQLParserException` will be thrown whenever a parser error, invalid number, date, or time occurs. Moreover, a `DriverException` will be thrown whenever an IO problem or other error occurs, such as accessing invalid column or table names. In order to make this test cleaner, these kind of exception will be omitted in the methods description.

GetInstance()

```
public static LitebaseConnection getInstance()
```

It creates a `LitebaseConnection` for the default application id, storing the database in the main secondary storage memory. This method avoids the creation of more than one instance with the same creator id, path, and thread, which would lead to performance and memory problems.

This is the same of doing

```
LitebaseConnection.getInstance(Settings.applicationId);
```

GetInstance()

```
public static LitebaseConnection getInstance(java.lang.String appCrid)
```

It creates a `LitebaseConnection` for the given creator id, which must be 4 characters long and may (or not) be the same one of your application's, storing the database in the main secondary storage memory. This method avoids the creation of more than one instance with the same creator id, path, and thread, which would lead to performance and memory problems. If the application id is not 4 characters long, a `DriverException` will be thrown.

getInstance()

```
public static LitebaseConnection getInstance(java.lang.String appCrid,
                                             java.lang.String params)
```

It creates a `LitebaseConnection` for the given creator id, which must be 4 characters long and may (or not) be the same one of your application's, and with the given connection parameter list. This method avoids the creation of more than one instance with the same creator id, path, and thread, which would lead to performance and memory problems. If the application id is not 4 characters long, a `DriverException` will be thrown.

The second parameter, `params`, can be the path where it is desired to access the database. It must be an absolute path, otherwise a `DriverException` will be thrown. If `null` is passed as this parameter, the default folder will be used. For Java and Blackberry, the default path is *Settings.data*`Path`. For Palm OS, the default is `/Litebase_DBs/`. Finally, for the other systems, the default path is given by *Settings.*`appPath`.

To use ASCII tables, the parameters must have the format:

`“chars_type = type; path = path_name”`

where `chars_type` can be `ascii` to store ASCII strings or `unicode` to use Unicode strings in the tables, and `path` has the same format of the preceding case. It is not possible to use just one parameter. If only the path is passed, the default format is Unicode. This is also the format used for the above versions of this method.

Note: Although `getInstance()` is a singleton, it is not recommended to call this method more than once for the same connection in the same thread. This will only make your application inefficient due to many method calls. It is much better to hook the connection with Litebase in a static variable or to pass it as a parameter.

A very important thing to be noticed is that two different connections will be created on Java SE and on Blackberry if one of the `getInstance()` calls is in the constructor of the `MainWindown` or in a method called by it and the other call is in a `initUI()`, `onEvent()`, or in its sub-methods.

getSourcePath()

`public java.lang.String` **`getSourcePath`**`()`

Returns the source path used by the Litebase connection. This path is where the tables of this connection are stored. This path won't return the slot used on palm. To see the slot being used, use `LitebaseConnection.getSlot()`.

execute()

`public void` **`execute`**`(java.lang.String sql)`

Used to execute a *create table* or *create index* SQL commands.

The index can be created after data was added to the table.

An `AlreadyCreatedException` may be thrown if the index/table was already created.

The following commands are supported:

- “CREATE TABLE *table_name* ({*column_name column_data_type*[(*type_size* [multiplier])] [*PRIMARY KEY*] [DEFAULT *value_default*] [NOT NULL] } [,...] [, PRIMARY KEY (pk_column_name [,...])])”

Description

- Creates the table with the given column definitions.

Parameters

- table_name: the name of the table to be created.
- column_name: the name of the columns to be created. One or more. Cannot have space, neither has a size limit.
- column_data_type: the data type of the column. Must be one of the following (case insensitive):
  - SHORT: represents the Java *short* type. Range from -32768 to +32767.
  - INT: represents the Java *int*eger type. Range from -2.147.483.648 to +2.147.483.647.
  - LONG: represents the Java *long* type. Range from -9.223.372.036.854.775.808 to +9.223.372.036.854.775.807.
  - FLOAT: represents the Java *float* type. Range from -3,40292347E+38 to +3,40292347E+38 on iPhone and from 1.4E-45 to +3,40292347E+38 on the other platforms.
  - DOUBLE: represents the Java *double* type. Range from 2.2250738585072014E-308 to +1,79769313486231570E+308 on iPhone and from 4.9E-324 to +1,79769313486231570E+308 on the other platforms..
  - CHAR: represents a case sensitive char array (each char range u0000' to '\uFFFF'). This is the one column type that must have the *type size* specified, from 1 to 65535. E.G.: char(1), char(2000), etc.
  - CHAR NOCASE: equivalent to CHAR, but searches are **case insensitive,** making the searches slower. E.G.: CHAR (20) NOCASE.
  - VARCHAR: internally it is equivalent to CHAR.
  - DATE: represents the Java/TotalCross *Date* type. Ranges from all valid dates between 1000/01/01 and 2999/12/31.
    - A date 'YYYY/MM/DD' (YYYY = year, MM = Month, DD = Day) is stored as a integer YYYYMMDD.
  - DATETIME: represents the Java/TotalCross Time type. Ranges from all possible dates and times with valid dates. It is stored as two integers.
    - the first integer represents a date 'YYYY/MM/DD' (YYYY = year, MM = Month, DD = Day) and is stored as a integer YYYYMMDD.
    - the second, a time 'HH:MM:SS:ZZZ' (HH = hour, MM = minutes, SS = seconds, ZZZ = millis) and is stored as a integer HHMMSSZZZ.
  - BLOB: represents an array of bytes. It can be a picture or a video, for example. It has also a multiplier, which can be K (kilo) or M (mega).

Its total size cannot be greater than 10 Mb.

- type_size: used only for the CHAR, CHAR NOCASE, VARCHAR and BLOB column types. See above.
- primary key: determines that this column is the primary key one (only for one column of the table).
- value_default: the default value for the field. It must be of the same type of the field.
- NOT NULL indicates that the column can't have a null.
- pk_column_name: the column names to specify a composed primary key.

Examples

```
CREATE TABLE person (name char(40) nocase primary key,birthday int default 20 not null,salary double default 2325.00, gender char(1),married char(3), age short)
```

```
CREATE TABLE person2 (name char(40) nocase,birthday int default 20 not null,salary double default 2325.00, gender char(1),married char(3), age short, primary key (name, birthday))
```

More about DATE and DATETIME

- Use quotes for this data type.
- White spaces in the beginning or in the end are not considered.
- The date must be inserted in the one of following formats:
  - DATEFORMAT
    - “YYYY/MM/DD”, “YY/MM/DD”, “YY-MM-DD”, “YY-MM-DD”
- The DATETIME must be inserted in one of the following formats:
  - DATEFORMAT + space + TIMEFORMAT
  - TIMEFORMAT
    - “HH:MM:SS:ZZZ”, “HH:MM:SS”, “HH:MM”, “HH”, “H”
- The select statements recoveries DATE and DATETIME according with `appSettings`.

Examples

```
create table person(name char(16), age int, birth date primary key, years DateTime)
```

```
insert into person values ('renato novais', 12, '2005/9-12', '2006/08-21 12:08')
```

```
insert into person values ('indira gomes', 13, '  2005/9-12', '05-4/3 1:8:59'

insert into person values ('Zenes Oliveira', 20, '07/9-13', '2006/08-21   13:08:59:431  '

Select name, age, birth, years from person

Select name, age, birth, years from person where birth = ' 2005/09/12'

Select name, age, birth, years from person where years != '05-4/3 01:8:59  '

Select min(years) as abirth from person

Select max(birth) as abirth from person
```

- "CREATE INDEX *index_name* ON *table_name*(*column_name* [,...])"

  Description

  - Creates an index for the given table column.

  Parameters

  - index_name: the name of the index.
    Obs.: The index_name is ignored and formed internally (but must be included in the SQL command).
  - table_name: the name of table.
  - column_name: the name of column to be created the index. Here you can specify one or more column names. Two or more column names generates a composed index.

  Examples

```
CREATE INDEX indexperson ON person(birthday)

CREATE INDEX indexperson2 ON person(name, birthday)
```

executeUpdate()

```
public int executeUpdate(java.lang.String sql)
```

Used to execute updates in a table (insert, delete, update, alter table, drop).

The following commands are supported:

- “ALTER TABLE *table_name* DROP primary key”

  Description

  - Deletes the primary key of a table.

  Parameters

  - table_name: the name of the table.

  Throws

  - DriverException: if the table does not have a primary key.

  - Example

```
ALTER TABLE person DROP primary key
```

- “ALTER TABLE *table_name* ADD primary key(*column_name* [,...])”

  Description

  - Adds a primary key to a table.

  Parameters

  - table_name: the name of the table.
  - column_name: the name(s) of the column(s) to be created the primary key.

  Throws

  - PrimaryKeyViolationException: if there is a repeated or null key in the table.

  Example

```
ALTER TABLE person ADD primary key (salary)
```

```
ALTER TABLE person ADD primary key (name, salary)
```

- “ALTER TABLE *table_name* RENAME TO *new_table_name*”

  Description

  - Renames a table from *table_name* to *new_table_name.*

  Parameters

  - table_name: the name of the table to be changed.
  - new_table_name: the new name of the table.

  Throws

  - DriverException: if the new table name already exists.

  Example

  ```
  ALTER TABLE person RENAME TO employee
  ```

- “ALTER TABLE *table_name* RENAME *column_name* TO *new_column_name*”

  Description

  - Renames a column of a table from *column_name* to *new_column_name.*

  Parameters

  - table_name: the name of the table where the column will be changed.
  - column_name: the name of the column to be changed.
  - new_column_name: the new name of the column.

  Throws

  - DriverException: if the new table already exists.

  Example

  ```
  ALTER TABLE person RENAME age TO years
  ```

- “DROP TABLE *table_name*”

Description

- Deletes the table and all associated indices.

Parameters

- table_name: the name of the table.

Example

```
DROP TABLE person
```

- “DROP INDEX {column_name [,...] | *} on *table_name*”

Description

- Deletes an index from the given table name. The index can be specified by its name(s) of the correspondent column(s) or by a wild card symbol (*), which deletes all indices associated with the table. ATTENTION: The index associated to the primary key is not deleted. So, if you want to delete this one you must use the ALTER TABLE statement.

Parameters

- column_name: the name of the column. You can specify one or more fields name.
- * : specifies to delete all indices associated to the table, except the primary key index.
- table_name: the name of the table.

Examples

```
DROP INDEX salary ON person
```

```
DROP INDEX * ON person
```

```
DROP INDEX name, salary ON person
```

- “INSERT INTO *table_name* [(*column_name* [, ...])] { VALUES ({ *values* } [, ...]) }”

Description

- Inserts the given row into the table. The target column names may be listed in any order. The driver assume DEFAULT values if you declared it. You can insert NULL too, but only if you didn't declare the field as NOT NULL. Note that you must give the column names only if you change their declared order. If no list of column names is given, the default is all the columns of the table in their declared order.

Parameters

- table_name: the name of the table.
- column_name: the name of column (remember that you can specify either ALL or ANY).
- values: a value to assign to the corresponding *column.*

Throws

- PrimaryKeyViolationException: if there is a repeated or null key in the table.

Examples

```
INSERT INTO person (region, age, salary, married, birthday, gender, name) VALUES (1, 25, 10000, 'no', 19790606, 'M', 'Renato Novais')
```

```
INSERT INTO person VALUES ('Indira Gomes', 19810228, 6000, 'F', 'no', 20, 2)
INSERT INTO person VALUES ('Caculé', 19800806, 5000, 'M', 'yes',25, 5)
```

```
INSERT INTO person(name, age) VALUES ('Caio', null)
// Fields region, salary, married, birthday, and gender will be
// filled with default values (if you declared them) or null value
// if permitted.
```

- “UPDATE *table_name* SET *column_name* = {*expression*} [,...]
  [WHERE *condition*] ”

Description

- Changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need to be mentioned in the SET clause; columns not explicitly modified retain their previous values. You can specify the condition in WHERE clause to determines what rows will be modified.

Parameters

- table_name: the name of the table.
- column_name: the name of the column.
- expression : an expression to assign to the column.
- condition: an expression that returns a value of type boolean. Only rows for which this expression returns true will be updated.

Throws

- PrimaryKeyViolationException: if there is a repeated or null key in the table.

Example

```
UPDATE person SET salary = 8000 WHERE age = 20 or age is null
```

- "DELETE FROM *table_name* [WHERE *condition*]"

Description

- Deletes a row or a set of rows in agreement with *conditions* in WHERE clause. If *condition* is omitted, the whole table is deleted, but its structure is preserved. The key word FROM is optional.

Parameters

- table_name: the name of the table.
- condition: an expression that returns a value of type boolean. Only rows for which this expression returns true will be deleted.

Examples

```
DELETE FROM person WHERE name = 'Caculé'
```

```
DELETE person WHERE age is not null
```

executeQuery()

`public ResultSet **executeQuery**(java.lang.String sql)`

Used to execute queries in a table. The whole table is searched for records that satisfies the conditions of the where clause, and the result is stored in memory. When

the `ResultSet` is closed, the memory is released. So its always a good idea to close the result set as soon as its no longer needed.

The following commands are supported:

- "SELECT * | *column_name* [AS *output_name*] [, ...] | function_name(*column_name)* [AS *output_name*] [, ...] FROM *table_name* [[AS] *table_name_alias*] [,...] [WHERE *condition*] [GROUP BY *column_name* [, ...]] [HAVING *condition*] [ORDER BY *column_name* [ASC | DESC] [, ...]]"

Description

- SELECT retrieves rows from one or more tables. The general processing of SELECT is as follows.

Parameters

- * : use this to return all columns in a query.
- column_name: here you specify the column names to be retrieved, grouped or ordered.
- function_name: the function applied on column_name. Function_name must be on of the Litebase functions.
- table_name: the name(s) of the table(s) that will be retrieved

Examples

```
SELECT * FROM person
```

```
SELECT * FROM person, department
```

IMPORTANT OBSERVATIONS

- `SELECT * from table_name`

  If a query of this format is issued, with a wild card, a single table, no where, order by, or group by clauses, the query processing is faster since the table is not analyzed. Therefore, if `table_name` is altered, the result set will be changed if it is already opened. For instance, if this kind of select is issued, not closed and a new row is inserted, the result set will also return the new row.
- Whenever possible, do not use where clauses that do nothing such as 1=1. This increase parser time and the where clause will not be all resolved using indices, turning the query processing slower.
- If instead of a `*` the user lists all the fields of `table_name` in its correct order

beginning with rowid, the query processing is as faster as when using *. However, if an alias is used, it will be ignored in `ResultSetMetaData`.

- Queries with no sorting, join and aggregations do not use temporary tables. If the table is altered and a result set with not temporary table is still opened, it will be inconsistent.
- Condition clause: WHERE
  - LIKE Statement
    - The char '%' is a wild card in SQL. You can use '%' at the start, at the end, both, in the middle (just one), or don't use it. In the last case, it is the same as equals (=).
    - Examples: WHERE <column_table>...

      `LIKE '%caculé'`: it returns the tuples that ends with 'caculé'.

      `LIKE 'renato%'`: it returns the tuples that starts with 'renato'.

      `LIKE 'indira%mes'`: it returns the tuples that starts with 'indira' and ends with 'mes'.

      `LIKE 'Zenes'`: it returns the tuples that is just equal to 'Zenes'.

      `LIKE '%Jener%'`: it returns the tuples that contains 'Jener'.

      `LIKE '%Luc%as%'`: it returns the tuples that contains 'Luc%as'.

      `LIKE 'ind%ira%mes'`: it returns the tuples that starts with 'ind' and ends with 'ira%mes'.

  - Order clause: ORDER BY
    - ASC is default.
    - You can put various columns in ORDER BY. In previous versions of Litebase before 2.00, there would be no effect.
    - In previous versions of Litebase, the DESC expression would only work if you used it on the first field. In version 2.00 and above, it works correctly.
    - Examples

`ORDER BY name, age`: orders by name and by age.

`ORDER BY name, age DESC`: orders by name in ascending order and age in descending order.

`ORDER BY name DESC, age`: orders by name in descending order and age in descending order.

- Aggregation functions: GROUP BY
  - An alias is required for the aggregate function column
  - The following aggregation functions are permitted:
    - avg
    - max
    - min
    - sum
    - count(*)

  - Example

```
ResultSet rs = null;
try
{
    rs = pdb.executeQuery("select age, sum(region) as tot from
                                        person group by age");
}catch(Exception e)
{
   Vm.debug(e.getMessage());
}
while (rs.next())
{
   Vm.debug("Name: " + rs.getString("age")
          + ", Total_Region: " + rs.getString("tot"));
}
```

    - Note that the alias “tot” must be used in the result set.
    - A `ResultSet` object is returned, as shown in the example above, and can be used to traverse the rows and also to get the column values.
    - When the column is of type LONG, you must append a suffix L (or

l) to the number. E.G.: 20050411095951L. Also, when using FLOAT types, you must suffix it with a F (or f), otherwise the default type will be DOUBLE.

- There's a special column name, **rowid**, that can be retrieved or used in the condition.
  - rowid cannot be updated. It is automatically generated for each row in the table, and is unique for each row, even after row deletions.
  - If you use the rowid in a condition clause, it is strongly suggested that you create an index for it.
  - When using *, the rowid is not returned! You must explicitly ask for it, followed by the other columns you want to retrieve.
- The `ResultSet` class contains a very useful method that returns a string matrix with all fields in the `ResultSet`. It speeds up the filling of `ListBox` and `Grid`. All selected columns are transformed to string.
- You can use the `setDecimalPlaces()` to define how the doubles will be formatted when being converted to string.

- More Examples:

```
ResultSet rs = driver.executeQuery("select rowid, name,
                   salary, age from person where age != 44");
rs.afterLast();
while (rs.next())
{
   Vm.debug("RowID: " + rs.getString(1) +
   " Name: " + rs.getString(2) + " Salary: " +
   rs.getString(3) + " - " + rs.getInt("age")+ " years" );
}


// A method that returns a string matrix to fill a ListBox
or a Grid.
public String[][] search(char startChar)
{
   String sql;

   // the select statements return only rows where name
   // starts with 'startChar'.
   sql = "SELECT rowid, name, gender, married, age, salary
          FROM person WHERE name like '" + startChar + "%'";
   ResultSet rs = pdb.executeQuery(sql);
   if (rs.first())
```

```
    {
       return rs.getStrings(-1,true,false);
    }
    else
    {
       return null;
    }
}

// Using the method 'seach'.
private Grid gridNames;

// The initialization of the Grid must be implemented.

String[][] people = search("R");
if (people == null)
{
   gridNames.clear();
   Vm.debug("There's no person started with this char");
}
else
{
   gridNames.setItems(people);
}
```

prepareStatement()

`public PreparedStatement` **`prepareStatement`**`(java.lang.String sql)`

Creates a pre-compiled statement with the given SQL. Prepared statements are faster for repeated queries; instead of parsing the same query where only a few arguments change, you create a prepared statement and the query is pre-parsed. Then, you just set the arguments (defined as `?` in the SQL) and run the SQL, where the index for the argument starts at 0. Note that recreating and recompiling the prepared statement with the same SQL is a waste of time, which reduces the performance of your program.

It is possible to find a complete example in `PreparedStatement` class section.

When a table is dropped and recreated, the prepared statements that use it must be re-prepared because the re-created table might have a different structure. This must also be done when the connection is closed and re-opened again. The explanation for this is similar: the new connection may be different and Litebase may behave unecpectly if this could be done.

getCurrentRowId()

`public int` **`getCurrentRowId`**`(java.lang.String tableName)`

Returns the current rowid for a given table. If the table doesn't exists, a

`DriverException` is thrown.

This method is useful to find the rowid of the last inserted record: just subtract 1 from the returned value. After a record is inserted, the rowId is incremented, so, if you get the rowid after you insert the record, you must subtract 1; if you get the rowid before inserting the row, then that's exactly the value.

Parameters:

- tableName: the name of the table.

Returns:

The current rowid.

Only note that (although very difficult to happen) the rowid may be < 0, because it is internally stored in the last 28 bits of an *unsigned int32*, and, thus, if the rowid gets over 134,217,728, it will be returned as a negative number in this method. But this causes no harm. It can be stored as a negative value that it will be correctly used by Litebase. Moreover, it is very unlikely that a table for nowadays devices will get more then this huge number of inserts. However, if so, please re-create your table from scratch.

getRowCount()

`public int` **`getRowCount`**`(java.lang.String tableName)`

Returns the number of valid rows in a table. This may be different from the number of records if a row has been deleted.

Parameters:

- tableName: The table name to get number of rows.

Returns:

The number of rows.

See Also:

`getRowCountDeleted().`

setRowInc()

`public void` **`setRowInc`**`(java.lang.String tableName, int inc)`

Sets the row increment used when creating or updating big amounts of data.

Using this method greatly increases the speed of bulk insertions (about 3x faster). To use it, you must call it (preferable) with the amount of rows that will be inserted.

After the insertion is finished, you **MUST** call it again, passing -1 as the inc argument. Without doing this last step, you may lose data because some writes will be delayed until you call it with -1. Another good optimization on bulk insertions is to drop the indices and then create them afterwards. So, to correctly use `setRowInc()`, you must::

```
driver.setRowInc("table", totalNumberOfRows);
// fetch the data and insert them.
driver.setRowInc("table", -1);
```

Using prepared statements on insertion makes it another couple of times faster.

convert()

`public boolean convert(java.lang.String tableName)`

Converts a table from the previous Litebase table version to the current one. If the table format is older than the previous table version, this method can't be used and a `DriverException` will be thrown. It is possible to know if the table version is not compatible with the current version used in Litebase because an exception will be thrown if one tries to open a table with the old format. The table name to be converted must be specified. The table will be closed after using this method and must be closed before it. Notice that the table .db file will be overwritten, so it is highly recommended to backup the tables. Nowadays, the `convert()` works to convert from versions 2.1 till 2.14 to 2.30. Tables from the versions 2.20 to 2.28 do not need to be converted to the table format for Litebase 2.30.

For more information of how to use this, see the Migration section.

exists()

`public boolean exists(java.lang.String tableName)`

Returns if the given table already exists. You can use this method before drop table.

Example:

```
if (pdb.exists("person")
{
   pdb.executeUpdate("DROP TABLE person");
}
```

closeAll()

`public void closeAll()`

Closes all files, so that they are immediately written to disk (on the desktop) or releases the file handles (on the device). When this method is issued, all tables

opened in the connection are also closed and the prepared statements will be in an invalid state. Therefore, the prepared statement of the closed connection must be prepared again if they will be used later on. Note that, after this is called, all result sets and prepared statements created with this Litebase instance will be in an inconsistent state, and using them will raise an `IllegalStateException`. This method also deletes the active instance for this creator id from our internal table.

purge()

`public int` **`purge`**`(java.lang.String tableName)`

Used to delete physically the records of the given table. Records are always deleted logically, to avoid the need of recreating the indices. When a new record is added, it doesn't occupy the position of the previously deleted one. This can make the table big, if you create a table, fill it and delete a couple of records without ever adding others, thus wasting space. This method will remove all deleted records and recreate the indices accordingly. Note that it can take some time to run. Therefore this operation should not be executed very often, This should not be done if your table has less then 10% of deleted rows.

Important: the rowid of the records is NOT changed with this operation.

Parameters:

- tableName: The table name to purge.

Returns:

The number of purged records.

getRowCountDeleted()

`public int` **`getRowCountDeleted`**`(java.lang.String tableName)`

Returns the number of deleted rows.

getRowIterator()

`public RowIterator` **`getRowIterator`**`(java.lang.String tableName)`

With it you can iterate through all the rows of a table in sequence and get its attributes. This is good for synchronizing a table. While the iterator is active, you must not do any queries or updates, unless you want to corrupt your data.

Parameters:

- tableName: The table name to get a row Iterator.

Example:

```
// This method removes physically all deleted records on table
// person and sets all records as sync.
public void setTableSync()
{
   RowIterator ri = null;
   try
   {

      // Delete physically the rows
      pdb.purge("person");

      // Set all as sync.
      ri = pdb.getRowIterator("person");

      while (ri.next())
      {
         ri.setSynced();
      }

   } catch (Exception ex)
   {
      \* ... *\
   }
   finally
   {
      if (ri != null)
      {
         ri.close();
      }
   }
}
```

getLogger()

public static synchronized `Logger getLogger()`

Gets the current Litebase logger.

`LitebaseConnection.logger` is now public and now can be accessed directly. This should be done unless logger is used within threads.

setLogger()

public static synchronized void `setLogger(Logger logger)`

Sets the litebase logger. This enables log messages for all queries and statements of Litebase and can be very useful to help finding bugs in the system. Logs take up memory space, so turn them on only when necessary.

`LitebaseConnection.logger` is now public and now can be accessed directly. This should be done unless logger is used within threads.

getDefaultLogger()

public static synchronized Logger getDefaultLogger()

Gets the default Litebase logger. This was already explained.

DeleteLogFiles()

public static int **deleteLogFiles**()

Deletes all log files found in the device. If log is enabled, the current log file is not affected by this command.

Returns:

the number of files deleted.

processLogs()

public static LitebaseConnection **processLogs**(String[] sql, String params, boolean debug)

This is a handy method that can be used to reproduce all commands of a log file. This is intended to be used by the development team only.

Here's a sample of how to use it:

```
String []sql =
{
   "new LitebaseConnection(MBSL,null)",
   "create table PRODUTO (IDPRODUTO int, IDPRODUTOERP char(10),
    IDGRUPOPRODUTO int, IDSUBGRUPOPRODUTO int, IDEMPRESA char(20),
    DESCRICAO char(100), UNDCAIXA char(10), PESO float,
    UNIDADEMEDIDA char(3), EMBALAGEM char(10), PORCTROCA float,
    PERMITETROCA int)",
   "create index IDX_PRODUTO_1 on PRODUTO(IDPRODUTO)",
   "create index IDX_PRODUTO_2 on PRODUTO(IDGRUPOPRODUTO)",
   "create index IDX_PRODUTO_3 on PRODUTO(IDEMPRESA)",
   "create index IDX_PRODUTO_4 on PRODUTO(DESCRICAO)",
   "closeAll",
   "new LitebaseConnection(MBSL,null)",
   "insert into PRODUTO values(1, '19132', 2, 1, '1, 2, 3',
    'ABSORVENTE SILHO ABAS','5',13,'PCT','20X30',10,0)"
};
LitebaseConnection.processLogs(sql, true);
```

It returns the `LitebaseConnection` instance created, or `null` if the `closeall()` was the last command executed (or no commands were executed at all).

Set `debug` to true if the log is to be debugged. Finally, for this method, the path can't be passed in `new LitebaseConnection()`. `params` must be used instead. It is not forbidden to pass a path to the connection, but it is not used.

recoverTable()

public boolean **recoverTable**(String tableName)

If a table is not closed properly, that is, closed when exiting the application properly or issuing a `LitebaseConnection.closeAll()`, when trying to open it again, a `TableNotClosedException` will be thrown. This does not mean that your table is corrupted, but it can be the case.

Therefore, whenever a table cannot be opened because of the exception above, `LitebaseConnection.recoverTable()` should be used to recover it. Notice that the table must be closed before using it, which will be the case if a `TableNotClosedException` is thrown. It will also be closed after this method finishes.

This method will invalidate every record whose CRC does not match the stored CRC, marking it as deleted and settings its rowid to zero. The corrupted records can be accessed via `RowIterator` methods. If the table was closed properly before calling this method, a `DriverException` will be thrown because there is nothing to be recovered.

getSlot()

`public int **getSlot**()`

This method returns the slot being used on palm. On the other devices, it return -1.

isOpen()

`public boolean **isOpen**(String tableName) throws DriverException, NullPointerException`

This method returns true if the table named `tableName` is opened in the current connection; false, otherwise.

Parameters:

- tableName: The table name to get a row Iterator.

Returns:

- true if the table named `tableName` is opened in the current connection; false, otherwise.

dropDatabase()

`public static void **dropDatabase**(String crid, String sourcePath, int slot)`

Drops all the tables from a database represented by its application id and path.

Parameters:

- crid: The application id of the database.
- sourcePath: The path where the files are stored.
- slot: The slot on Palm where the source path folder is stored. Ignored on other

platforms.

Throws:

- DriverException: If the database is not found.

RESULTSET CLASS

This class represents a set or rows resulting from a `LitebaseConnection.executeQuery()` method call. It cannot be directly instantiated. With this class you can cover the "virtual rows" of table, and you can discover the meta data of a table. These methods will also throw an `IllegalStateException` if the result set or the Litebase connection where it was created is closed. IO problems will also raise `DriverException`s. When accessing the result set columns, invalid indices will raise an `IllegalArgumentException`, unknown column names will raise a `DriverException`, and trying to fetch a column value from the result set with a method to fetch data with a incompatible type will result in `DriverException`.

getResultSetMetaData()

`public ResultSetMetaData` **`getResultSetMetaData`**`()`

Returns the meta data for this result set. See the `ResultSetMetaData` class for more information. There you will find a good example with `ResultSet` and the `ResultSetMetaData` classes.

close()

`public void` **`close`**`()`

Releases all memory allocated for this object. Its a GOOD idea to call this when you no longer need it, but it is also called by the GC when the object is no longer in use. This can be issued even if the driver is already closed.

beforeFirst()

`public void` **`beforeFirst`**`()`

Places the cursor before the first record.

afterLast()

`public void` **`afterLast`**`()`

Places the cursor after the last record.

first()

`public boolean` **`first`**`()`

Places the cursor in the first record.

last()

`public boolean last()`

Places the cursor in the last record.

next()

`public boolean next()`

Returns the next record of this `ResultSet`.

prev()

`public boolean prev()`

Returns the previous record of this `ResultSet`.

Throws:

`DriverException` - If an error occurs.

getShort()

`public short getShort(int col)`

Given the column index (starting from 1), returns a short value that is represented by this column. Note that you must only request this column as short if it was created with this precision or if the data being fetched is the result of a DATE or DATETIME SQL function. If the value is SQL `null`, the value returned is 0.

getShort()

`public short getShort(java.lang.String colName)`

Given the column name (case insensitive), returns a short value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getInt()

`public int getInt(int col)`

Given the column index (starting from 1), returns an int value that is represented by this column. Note that you must only request this column as int if it was created with this precision. If the value is SQL `null`, the value returned is 0.

getInt()

`public int getInt(java.lang.String colName)`

Given the column name (case insensitive), returns an int value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getLong()

`public long getLong(int col)`

Given the column index (starting from 1), returns a long value that is represented by this column. Note that you must only request this column as long if it was created with this precision. If the value is SQL `null`, the value returned is 0.

getLong()

`public long getLong(java.lang.String colName)`

Given the column name (case insensitive), returns a long value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getFloat()

`public float getFloat(int col)`

Given the column index (starting from 1), returns a float value that is represented by this column. Note that you must only request this column as float if it was created with this precision. If the value is SQL `null`, the value returned is 0.0.

getFloat()

`public float getFloat(java.lang.String colName)`

Given the column name (case insensitive), returns a float value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getDouble()

`public double getDouble(int col)`

Given the column index (starting from 1), returns a double value that is represented by this column. Note that you must only request this column as double if it was created with this precision. If the value is SQL `null`, the value returned is 0.0.

getDouble()

`public double getDouble(java.lang.String colName)`

Given the column name (case insensitive), returns a double value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getChars()

`public char[] getChars(int col)`

Given the column index (starting from 1), returns a char array that is represented by this column. Note that you must only request this column as char[] if it was created as a string type (CHARS or CHARS_NOCASE). If the value is SQL `null`, the value returned is `null`.

getChars()

`public char[] getChars(java.lang.String colName)`

Given the column name (case insensitive), returns a char array that is represented by this column. It behaves similar to the previous one. However, it is slower.

getBlob()

`public byte[] getBlob(int col)`

Given the column index (starting from 1), returns a byte array that is represented by this column. Note that you must only request this column as byte[] if it was created as a blob. If the value is SQL `null`, the value returned is `null`.

getBlob()

`public byte[] getBlob(java.lang.String colName)`

Given the column name (case insensitive), returns a byte array that is represented by this column. It behaves similar to the previous one. However, it is slower.

getString()

`public java.lang.String getString(int col)`

Given the column index (starting from 1), returns a string that is represented by this column. Any column type can be returned as a string except blob. Double/float values formatting will use the precision set with the `setDecimalPlaces()` method. Attention: if the field has DATETIME format, the millis is not returned. You can use the `getTime()` to get the millis. If the value is SQL `null` or the column is of type blob, the value returned is `null`.

getString()

`public java.lang.String getString(java.lang.String colName)`

Given the column name (case insensitive), returns a string that is represented by this column. It behaves similar to the previous one. However, it is slower.

getStrings()

`public java.lang.String[][] getStrings()`

Starting from the current cursor position (must use `first()`, `last()`, `prev()` or `next()`, **not** `beforeFirst()` or `afterLast()`), it reads all result set rows that's being requested from the current one. If the value is SQL `null` or the column is of

type blob, the value returned is `null`.

Throws:

DriverException: if the result set cursor is in an invalid position.

getStrings()

`public java.lang.String[][] getStrings(int count)`

This is the most powerful method of this class. Starting from the current cursor position (must use `first()`, `last()`, `prev()` or `next()`, **not** `beforeFirst()` or `afterLast()`), it reads the smallest number of rows between the requested amount and the number of rows from the current cursor position till the end of the result set. If the value is SQL `null` or the column is of type blob, the value returned is null.

The parameter `count` is the number of rows to be fetched, or -1 for all.

It returns a string matrix, where `String[0]` is the first row, and `String[0][0]`, `String[0][1]...` are the column elements of the first row. Returns `null` if there are no more elements to be fetched.

Double/float values will be formatted using the `setDecimalPlaces()` settings.

Throws:

DriverException: if the result set pointer is in an invalid position.

IllegalArgumentException: if the count argument is smaller than -1.

NOTE: To populate a grid, since your application won't be able to show all the rows at the same time if there are many of them, it is better to use this version of `getStrings()` instead of loading all the result set in the grid using the previous method. This way, the grid is being filled in runtime when the user rolls the scroll bar. Moreover, this reduces the chances of getting an `OutOfMemoryError`.

Example:

```
import litebase.*;
import totalcross.ui.*;
import totalcross.ui.dialog.*;
import totalcross.ui.event.*;
import totalcross.util.Comparable;
import totalcross.sys.*;

public class TCTestWin extends MainWindow implements PressListener,
```

```
Grid.DataSource
{
   static class CodDesc implements Comparable
   {
      String desc;
      int index; // The table is ordered by codigo.
      static ResultSet rs;

      public CodDesc(int idx)
      {
         index = idx;
         desc = rs.getString("descricao");
      }

      public int compareTo(Object other) throws ClassCastException
      {
         return desc.compareTo(((CodDesc)other).desc);
      }
   }

   LitebaseConnection lb = LitebaseConnection.getInstance();

   Grid grid;
   RadioGroupController rg;
   CodDesc[] cds; // Ordered by codigo.
   CodDesc[] dcs; // Ordered by descricao.
   Label stat;

   public TCTestWin()
   {
      setUIStyle(Settings.Vista);
   }

   public void initUI()
   {
      try
      {
         stat = new Label("Loading...");
         stat.setFont(font.asBold());
         add(stat, LEFT, AFTER, FILL, PREFERRED);
```

```
        repaintNow();

        // First, loads.
        int ini = Vm.getTimeStamp();
        ResultSet rs = lb.executeQuery("select rowid, codigo, codigogrupo,
descricao, embalagem, vendaminima, preco, codigobarras, estoque, obs,
flaggrade, falta, novo, promocao, descontocmaximo from tblafvproduto01");
        cds = new CodDesc[rs.getRowCount()];
        CodDesc.rs = rs;
        for (int i=0; rs.next(); i++)
           cds[i] = new CodDesc(i);
        stat.setText("Load time: " + (Vm.getTimeStamp() - ini) + "ms");
        stat.repaintNow();

        // Now, sorts.
        ini = Vm.getTimeStamp();
        dcs = new CodDesc[cds.length];
        Vm.arrayCopy(cds, 0, dcs, 0, cds.length); // Copies the references
        qsortDesc(dcs, 0, dcs.length - 1);
        stat.setText(stat.getText() + " . Sorting: "
                              +  (Vm.getTimeStamp() - ini) + "ms");

        String[] tit {"rowid", "codigo", "codigogrupo", "descricao",
 "embalagem", "vendaminima", "preco", "codigobarras", "estoque", "obs",
 "flaggrade", "falta", "novo", "promocao", "descontocmaximo"};
        int[] align = {LEFT, LEFT, LEFT, LEFT, LEFT, LEFT, LEFT, LEFT,
                           LEFT, LEFT, LEFT, LEFT, LEFT, LEFT, LEFT};

        // Hides rowid and codigogrupo. Negative values mean %.
        int[] widths = {0, -10, 0, -30, -10, -10, -10, -10, -10, -10,
-10, -10, -10, -10, -10};

        Radio r = new Radio("Código", rg = new RadioGroupController());
        add(new Label("Order by"), LEFT, AFTER, PREFERRED,
                                          r.getPreferredHeight());
        add(r, AFTER + 2, SAME); r.addPressListener(this);
        r.setChecked(true);
        add(r = new Radio("Descrição", rg), AFTER + 2, SAME);
        r.addPressListener(this);
        Grid.useHorizontalScrollBar = true;
        add(grid = new Grid(tit, widths, align, false), LEFT, AFTER + 2,
                                                  FILL, FILL);
```

```
      Grid.useHorizontalScrollBar = false;
      grid.liveScrolling = true;
      grid.setDataSource(this, cds.length);
   }
   catch (Exception e)
   {
      MessageBox.showException(e, false);
   }
}
public void controlPressed(ControlEvent e)
{
   grid.fetchDataSource();
}

public String[][] getItems(int startingRow, int count)
{
   ResultSet rs = CodDesc.rs;
   CodDesc[] qual = rg.getSelectedIndex() == 0? cds : dcs;

   // Initialies the array of strings that will store each row.
   String[][] v = new String[count][];
   for (int i = 0; i < count; i++)
   {
      CodDesc cd = qual[startingRow++];
      rs.absolute(cd.index);
      v[i] = rs.getStrings(1)[0];
   }
   return v;
}

static void qsortDesc(CodDesc[] items, int first, int last)
{
   if (first >= last)
      return;
   int low = first;
   int high = last;

   CodDesc mid = items[(first + last) >> 1];
   while (true)
   {
```

```
            String s = mid.desc;
            while (high >= low && s.compareTo(items[low].desc)  > 0)
               low++;
            while (high >= low && s.compareTo(items[high].desc) < 0)
               high--;
            if (low <= high)
            {
               CodDesc temp = items[low];
               items[low++] = items[high];
               items[high--] = temp;
            }
            else break;
         }

         if (first < high)
            qsortDesc(items, first, high);
         if (low < last)
            qsortDesc(items, low, last);
      }
   }
```

getDate()

`public Date getDate(int col)`

Given the column index (starting from 1), returns an Date value that is represented by this column. Note that you must only request this column as Date if it was created with Date data type. If the value is SQL `null`, the value returned is `null`.

getDate()

`public Date getDate(java.lang.String colName)`

Given the column name (case insensitive), returns an Date value that is represented by this column. It behaves similar to the previous one. However, it is slower.

getDateTime()

`public Time getDateTime(int col)`

Given the column index (starting from 1), returns a `Time` (corresponding to the DATETIME Litebase data type) value that is represented by this column. Note that you must only request this column as DateTime if it was created with this data type. If the value is SQL `null`, the value returned is `null`.

getDateTime()

`public Time getDateTime(java.lang.String colName)`

Given the column name (case insensitive), returns a `Time` (corresponding to the DATETIME Litebase data type) value that is represented by this column. It behaves similar to the previous one. However, it is slower.

absolute()

`public boolean absolute(int row)`

Places this result set cursor at the given absolute row (starting from 0). This is the absolute physical row of the result set table. You usually use this method to restore the row at a previous row returned with the `getRow()` method.

Parameters:

- row: the row number of where the cursor will be placed.

relative()

`public boolean relative(int rows)`

Moves the cursor rows in distance. The value can be greater or lower than zero. It searches the position until finding the right row or the end or the beginning of the result set table.

getRow()

`public int getRow()`

Returns the current physical row of the table where the cursor is (starting from 0). Must be used with the absolute method.

setDecimalPlaces()

`public void setDecimalPlaces(int col, int places)`

Sets the number of decimal places that the given column (starting from 1) will have when being converted to string. Must be used for columns of type DOUBLE or FLOAT only.

Throws:

DriverException: if the value for decimal places is invalid (< -1 or > 40) or the column is not of type float or double.

getRowCount()

`public int getRowCount()`

Returns the number of rows of this result set.

Example:

```
ResultSet rs = null;
try
{
   rs = pdb.executeQuery("select salary,age from person");
}
catch (Exception exception)
{
   /*...*/
}

// Sets decimal places for salary that is double.
rs.setDecimalPlaces(1, 5);

while (rs.next())
   Vm.debug("salary: " + rs.getString("salary") + ", age: " +
                                                  rs.getShort("age"));

rs.first(); // Places in the first rows.
Vm.debug("Row: " + rs.getRow() + " salary: " + rs.getString("salary")
                              + ", age: " + rs.getShort("age"));

// Moves the cursor rows in distance relative a current position.
rs.relative(rs.getRowCount() - 1);

Vm.debug("Row: " + rs.getRow() + " salary: " + rs.getString("salary")
                              + ",  age: " + rs.getShort("age"));
rs.absolute(2); // Places the cursor in an absolute position.
Vm.debug("Row: " + rs.getRow() + " salary: "+rs.getString("salary")
                              + ", age: " + rs.getShort("age"));
rs.close(); // Closes the resultset.
```

isNull()

`public boolean isNull(int col)`

Indicates if this column has a null.

isNull()

`public boolean isNull(int colName)`

Indicates if this column has a null. It behaves similar to the previous one. However, it is slower.

PREPAREDSTATEMENT CLASS

Represents a SQL Statement that can be prepared (compiled) once and executed many times with different parameter values. It cannot be directly instantiated.

It must be noticed that the methods to set the parameters should be used with the right data type of the table column. Otherwise, a `DriverException` will be raised. The only exception is concerning `setString()`, which can be used to set any type except for blobs. A `DriverException` will also be thrown if an IO error occurs. A `SQLParseException` will occur whenever invalid SQL, dates, datetimes or invalid numbers are used. An `IllegalStateException` will be thrown whenever a closed driver or prepared statement is accessed, whereas a `IllegalArgumentException` will be raised if an invalid parameter index is used.

executeQuery()

`public ResultSet` **`executeQuery`**`()`

This method executes a prepared SQL query and returns its `ResultSet`.

Returns:

The `ResultSet` of the SQL statement.

Throws:

DriverException: If the SQL used to prepare the prepared statement is not a select or not all the parameters are defined.

executeUpdate()

`public int` **`executeUpdate`**`()`

This method executes a SQL INSERT, UPDATE or DELETE statement. SQL statements that return nothing such as SQL DDL statements can also be executed.

Returns:

The result is either the row count for INSERT, UPDATE or DELETE statements; or 0 for SQL statements that return nothing.

Throws:

DriverException: If the SQL used to prepare the prepared statement is a select or not all the parameters are defined.

setShort()

```
public void setShort(int index, short value)
```

This method sets the specified parameter from the given Java short value.

Parameters:

- index: the index of the parameter value to set, starting from 0.
- value: the value of the parameter.

setInt()

```
public void setInt(int index, int value)
```

This method sets the specified parameter from the given Java int value, similar to the above method.

setLong()

```
public void setLong(int index, long value)
```

This method sets the specified parameter from the given Java long value, similar to the above method.

setFloat()

```
public void setFloat(int index, double value)
```

This method sets the specified parameter from the given Java float value, similar to the above method.

setDouble()

```
public void setDouble(int index, double value)
```

This method sets the specified parameter from the given Java double value, similar to the above method.

setString()

```
public void setString(int index, java.lang.String value)
```

This method sets the specified parameter from the given Java String value, similar to the above method. Moreover, this method can be used for Date and DateTime field types, where dates must be set in the format YMD. Notice that a null string can't be set in a where clause, otherwise a `SQLParseException` will be thrown.

Example:

```
LitebaseConnection driver = LitebaseConnection.getInstance(creatorId);
```

```
driver.execute("CREATE TABLE company1(name CHAR(32), birth DATE,
                                          years DATETIME)");

ps = driver.prepareStatement("INSERT INTO company1 VALUES(?, ?, ?)");
ps.setString(0, "maria marlene");
ps.setString(1, "2006/05/02");
ps.setString(2, "2005/04/11 12:15:27:102");
```

setBlob()

`public void setBlob(int index, byte[] value)`

This method sets the specified parameter from the given Java byte[] value, which is a blob, similar to the above method. This method will throw a `SQLParseException` if one tries to set a blob type in a parameter in a where clause. That is, `setBlob()` can only be used with updates and inserts, never with deletes or selects.

setDate()

`public void setDate(int index, totalcross.util.Date value)`

This method sets the specified parameter from the given `totalcross.util.Date` value, similar to the above method. Again, a null date can't be set in a where clause.

Example:

```
LitebaseConnection pdb = LitebaseConnection.getInstance(creatorId);

driver.execute("CREATE TABLE company1(name CHAR(32), birth DATE)");

ps = driver.prepareStatement("INSERT INTO company1 VALUES(?, ?)");
ps.setString(0, "Ana Franscica");
Date d = new Date("2006/05/06", Settings.DATE_DMY);
ps.setDate(1,d);
```

ATTENTION:

You must use with care the constructor new `Date(<string_date>)`. Some devices can construct different dates, according to the device's date format. For example, the constructor `new Date("12/09/2006")`, depending on the device's date format, can generate a date like "12 of September of 2006" or "09 of December of 2006". To avoid this, use the constructor `new Date(<string_date>, totalcross.sys.Settings.DATE_XXX)` instead, where `totalcross.sys.Settings.DATE_XXX` is a date format parameter that must be one of `the totalcross.sys.Settings.DATE_XXX` constants.

setDateTime()

`public void setDateTime(int index, totalcross.util.Date date)`

This method sets the specified parameter from the given `totalcross.util.Date`, similar to the above method. Again, a null date can't be set in a where clause.

Additionally, to set null, a type cast to Date must be done because to remove ambiguity with the next method.

Date date will be stored as a DATETIME Litebase field with 0 in its time part.

Example:

```
LitebaseConnection driver = LitebaseConnection.getInstance(creatorId);
driver.execute("CREATE TABLE company1(name CHAR(32), years DATETIME)");
ps = driver.prepareStatement("INSERT INTO company1 VALUES(?, ?)");
ps.setString(0, "josé sebastião");
ps.setDateTime(1, new Date(20060506));
```

setDateTime()

`public void setDateTime(int index, totalcross.sys.Time time)`

This method sets the specified parameter from the given `totalcross.util.Time` value, similar to the above method. Again, a null datetime can't be set in a where clause. Additionally, to set null, a type cast to Time must be done because to remove ambiguity with the previous method.

Time time will be stored as a DATETIME Litebase field.

Example:

```
LitebaseConnection driver = LitebaseConnection.getInstance(creatorId);
driver.execute("CREATE TABLE company1(name CHAR(32), years DATETIME)");
ps = driver.prepareStatement("INSERT INTO company1 VALUES(?, ?)");
ps.setString(0, "josé sebastião");
ps.setTime(1, new Time(2006, 02, 01, 12, 15, 24, 352));
```

setNull()

`public void setNull(int index)`

This method sets the specified parameter to `null`. This can be used to set any column type as null. It must be just remembered that a parameter in a where clause can't be set to null.

Parameters:

- index: the index of the parameter value to set, starting from 0.

Example:

```
LitebaseConnection driver = LitebaseConnection.getInstance(creatorId);

driver.execute("CREATE TABLE company1(name CHAR(32), years DATETIME)");

ps = driver.prepareStatement("INSERT INTO company1 VALUES(?, ?)");
ps.setString(0, "josé sebastião");
ps.setNull(1);
```

clearParameters()

`public void clearParameters()`

This method clears all of the input parameters that have been set on this statement.

toString()

`public java.lang.String toString()`

Returns the SQL used in this statement.

Example:

```
PreparedStatement psInsert, psUpdate, psDelete, psSelect;
String sqlInsert, sqlUpdate, sqlDelete, sqlSelect;


// driver is the LitebaseConnection instantiated above.

// Call this method in the start of the application, after  Litebase has
   been initializated and after the table has been created.
public void preparedStatement()
{
   try
   {
      sqlInsert = "INSERT INTO person VALUES (?, ?, ?, ?, ?, ?, ?)";
      sqlUpdate = "UPDATE person SET name = ?, birthday = ?, salary = ?,
      gender = ?, married = ?, age = ?, region  = ? WHERE rowid = ?";
      sqlDelete = "DELETE FROM person WHERE rowid = ?";

      sqlSelect = "SELECT * FROM person WHERE age >= ?";
      psInsert = pdb.prepareStatement(sqlInsert);
      psUpdate = pdb.prepareStatement(sqlUpdate);
      psDelete = pdb.prepareStatement(sqlDelete);
      psSelect = pdb.prepareStatement(sqlSelect);
   }

   catch(Exception e)
   {

      /*...*/

   }
}


// This method fills the prepared statement and executes.

public void setPreparedStatement()
{
   try
```

```
    {
       psInsert.setString(0, "Zenes Novais");
       psInsert.setDate(1, “1980/05/03”);
       psInsert.setDouble(2, 3540.0);
       psInsert.setString(3, "F");
       psInsert.setString(4, "Yes");
       psInsert.setInt(5, 26);
       psInsert.setInt(6, 8);
       psInsert.executeUpdate();
       psUpdate.setString(0, "Jener Novais");
       psUpdate.setDate(1, “1979/05/03”);
       psUpdate.setDouble(2, 3800.0);
       psUpdate.setString(3, "M");
       psUpdate.setString(4, "Yes");
       psUpdate.setInt(5, 27);
       psUpdate.setInt(6, 8);
       psUpdate.setInt(7, 4);     // Updates the row of rowid = 4.
       psUpdate.executeUpdate();
       psDelete.setInt(0, 3);     // Drops the row of rowid = 3.
       psDelete.executeUpdate();
       psSelect.setInt(0, 25);    // Sets the age in the select.
       ResultSet rs = psSelect.executeQuery();

       while (rs.next)
       {

          Vm.debug(“Age: ”+rs.getShort(“age”));

       }
    }

    catch(Exception e)
    {

       /*...*/

    }
 }
```

RESULTSETMETADATA CLASS

This class returns useful information for the result set columns. It cannot be directly instantiated. An `IllegalStateException` will be thrown if the driver or result set are closed, whereas an `IllegalArgumentException` will be thrown if an invalid column index is used. A `DriverException` will be thrown if an IO error occurs.

getColumnCount()

`public int` **`getColumnCount`**`()`

Returns the number of columns for this result set.

getColumnDisplaySize()

`public int` **`getColumnDisplaySize`**`(int column)`

Given the column index (starting at 1), returns the display size. For chars, it will

return the number of chars defined; for primitive types, it will return the number of decimal places it needs to be displayed correctly. Returns -1 if the column type is blob.

getColumnLabel()

`public java.lang.String getColumnLabel(int column)`

Given the column index (starting at 1), returns the column name. Note that if an alias for the column is used, the alias will be returned instead.

getColumnType()

`public int getColumnType(int column)`

Given the column index (starting at 1), returns the column type. Its values can be:

`SHORT_TYPE`, `INT_TYPE`, `LONG_TYPE`, `FLOAT_TYPE`, `DOUBLE_TYPE`, `CHAR_TYPE`, `CHAR_NOCASE_TYPE`, `DATE_TYPE`, `DATETIME_TYPE`, and `BLOB_TYPE`.

Example:

```
// This method fills the attributes below with table meta data
// - columnsName (contains names of columns).
// - columnsLength (contain length of columns) for types different from
   char.
// For example INT/DOUBLE will have columnsLength = max value for the TYPE
   (e. g.: DOUBLE = 21)
// - columnsType (contains types of columns).
private void discoverMetaDataTable(String[] columnsName,
                                    int[] columnsLength, int[] columnsType)
{
   ResultSet rs = null;
   try
   {
      openLb(); // Gets the instance of Litebase.
      rs = lb.executeQuery("SELECT * FROM person WHERE rowid = 10000");
      ResultSetMetaData rsmd = rs.getResultSetMetaData();
      int numCols = rsmd.getColumnCount(); // Gets the number of cols.

      // Fills the parameters.
      for (int i = 1; i <= numCols; i++)
      {
         columnsName[i - 1] = rsmd.getColumnLabel(i);
         columnsType[i - 1] = rsmd.getColumnType(i);
         columnsLength[i - 1] = rsmd.getColumnDisplaySize(i);
```

```
            }
        }
        catch(Exception e)
        {
            handleException(e);
        }
        finally
        {
            if (rs != null)
              rs.close();
            closeLb(); // Closes Litebase.
        }
    }
```

getColumnTypeName()

`public String **getColumnTypeName**(int column)`

Given the column index (starting at 1), returns the name of the column type.

getColumnTableName()

`public int **getColumnTableName**(int column)`

Given the column index (starting at 1), returns the name of the table it came from.

getColumnTableName()

`public int **getColumnTableName**(String columnName)`

Given the column name or alias, returns the name of the table it came from. It is important to notice that if the query is a join and two fields in the result set have the same name, only the name of the first table of this column name will be returned.

Throws:

DriverException: if the column name is not found.

hasDefaultValue()

`public boolean **hasDefaultValue**(int columnIdx)`

Given the column index (starting at 1), indicates if a column of the result set has default value.

Throws:

DriverException: if the column does not have an underlining table column, such as `count(*)`.

hasDefaultValue()

`public boolean` **`hasDefaultValue`**`(String columnName)`

Given the column name or alias, indicates if a column of the result set has default value.

Throws:

DriverException: if the column name does not have an underlining table column, such as `count(*)`, or is not found.

isNotNull()

`public boolean` **`isNotNull`**`(int columnIdx)`

Given the column index (starting at 1), indicates if a column of the result set is declared as `not null`.

Throws:

DriverException: if the column does not have an underlining table column, such as `count(*)`.

isNotNull()

`public boolean` **`isNotNull`**`(String columnName) throws DriverException`

Given the column name or alias, indicates if a column of the result set is declared as `not null`.

Throws:

DriverException: if the column name does not have an underlining table column, such as `count(*)`, or is not found.

ROWITERATOR CLASS

Class used to iterate through the rows of a database. It can access some attributes from the row that ease the control of which row was changed, deleted, or is newer since a synchronization. It can also be used to access the values of the columns of the current row.

An iterator cannot be constructed directly; it must be created through the method `LitebaseConnection.getIterator()`.

An `IllegalStateException` will be thrown if the driver or row iterator are closed, whereas an `IllegalArgumentException` will be thrown if an invalid column index is used. A `DriverException` will be thrown if an IO error occurs or a get method is used with the wrong type.

Useful members:

data

`public byte[] data`

The data for the current row. The whole row is included.

rowid

`public int rowid`

The rowid for the current row.

attr

`public byte attr`

The attribute for this row. You must use the constants `ROW_ATTR_SYNCED`, `ROW_ATTR_NEW`, `ROW_ATTR_UPDATED`, or `ROW_ATTR_DELETED` to compare. It must be noticed that changing this attribute does not change the current row attribute. That is, this should be READ ONLY.

rowNumber

`public int rowNumber`

The number of the row. This must be READ ONLY. Changing it will corrupt your database.

Methods available:

next()

`public boolean next()`

Moves to the next record and fills the data members.

nextNotSynced()

`public boolean nextNotSynced()`

Moves to the next record with an attribute different of `ROW_ATTR_SYNCED`.

setSynced()

`public void setSynced()`

If the attribute is currently NEW or UPDATED, this method sets them to SYNCED. Note that if the row is DELETED, the change will be ignored. That is, deleted rows are always marked not synchronized.

close()

```
public void close()
```

Closes this iterator. You will then be able to do queries in a safe way.

reset()

```
public void reset()
```

Resets the counter to zero so that it is possible to restart to fetch records.

Example:
Note: You can find another example about this topic in `LitebaseConnection.getRowIterator()` above.

```
private static final char INSERTED = 'I';
private static final char DELETED = 'D';
private static final char UPDATED = 'U';
private static final char SYNC = 'S';

// Returns a char correspondent to RowIterator.attr

private char attrToChar(RowIterator ri)
{
   switch (ri.attr) // USE SWITCH
   {
      case RowIterator.ROW_ATTR_NEW:
         return INSERTED;
      case RowIterator.ROW_ATTR_UPDATED:
         return UPDATED;
      case RowIterator.ROW_ATTR_DELETED:
         return DELETED;
      case RowIterator.ROW_ATTR_SYNCED:
         return SYNC;
      default:
         return '-';
   }
}
```

getShort()

```
public short getShort(int column)
```

Fetches a short stored in column number `column`. Returns 0 if the column is null.

getInt()

```
public int getInt(int column)
```

Fetches an integer stored in column number `column`. Returns 0 if the column is null.

getFloat()

```
public double getFloat(int column)
```

Fetches a floating point number stored in column number `column`. Returns 0 if the column is null.

getDouble()

```
public double getDouble(int column)
```

Fetches a double precision floating point number stored in column number `column`. Returns 0 if the column is null.

getString()

```
public String getString(int column)
```

Fetches a string stored in column number `column`. Returns null if the column is null.

getDate()

```
public Date getDate(int column)
```

Fetches a date stored in column number `column`. Returns null if the column is null.

getDateTime()

```
public Time getDateTime(int column)
```

Fetches a datetime stored in column number `column`. Returns null if the column is null.

getBlob()

```
public byte[] getBlob(int column)
```

Fetches a blob stored in column number `column`. Returns null if the column is null.

isNull()

```
public boolean isNull(int column)
```

Indicates if the value stored in a column is null.

MIGRATION FROM DIFFERENT TABLE FORMATS

This section will guide you on table migration generated by a previous Litebase 2 table version to a newer and more powerful version. Currently, it migrates from the version used for Litebase from version 2.10 to 2.14 to the version used from version 2.30. With this change you will also receive these advantages:

- A more reliable database with less corrupt data.
- More features implemented.
- A faster Litebase which uses less memory.

This change will facilitate the addition of new SQL features. Litebase is being prepared to support even more features in the future. To migrate the old tables; instructions below.

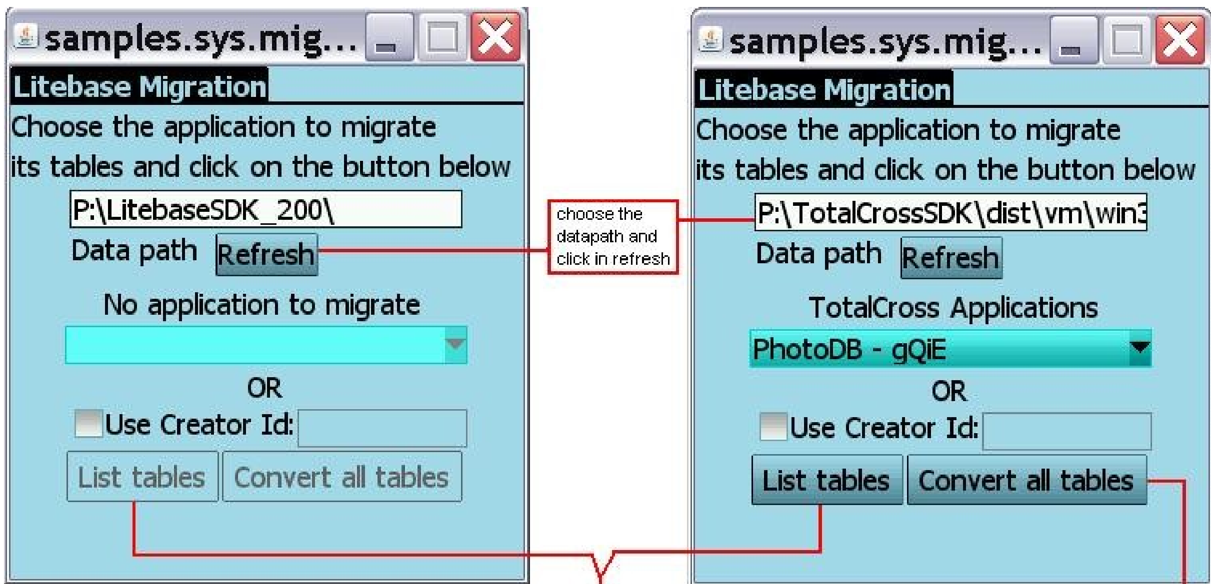
MIGRATING THE OLD TABLES

To migrate your old tables you have two choices.

1. Using the `convert()` method within `LitebaseConnection` class.

```
// Converting the table “person”.
LitebaseConnection driver = LitebaseConnection.getInstance(creatorId);
driver.convert(“person”);
```

2. Using a program sample (Figure 1) that is placed in `samples.sys.migration.*`. The appliaction files for each platform supported is on the folder `samples/sys/migration`. This program shows all the TotalCross applications that contain tables to be migrated (Figure 1.a) in the given folder. You can also use the creator id for which the table was created (Figure 1.b). Notice that in some platforms it is not possible to list the installed applications.



a. Using the Listbox applications

b. Using the creator Id

Convert all tables for the selected application or creator Id

Click on Migrate button to convert the selected tables

LITEBASE CONDUIT

PALM OS

Since HotSync does not support the installation of anything besides pdb and prc files, it's impossible to synchronize the Litebase files without a Conduit built specifically for this purpose.

Litebase Conduit (`samples.sys.tablecopier.LitebaseConduit`) uses the Conduit API to perform this task, allowing the installation and retrieval of Litebase files during the HotSync.

Requirements:

HotSync Manager version 6.0.1. (It may work with older or newer versions, but officially we support only this version of the HotSync Manager). For Windows Vista and 7, you can try using a newer HotSync Manager version or running version 6.0.1 on a XP emulator.

LitebaseConduit.prc must be installed on the device. The Litebase installer for PALM OS already includes this file.

WinCE and Windows Mobile

Unlike PALM OS devices, you can manually install and retrieve the Litebase files from WinCE and Windows Mobile devices. Litebase Conduit automates this task, synchronizing the Litebase files every time the device is connected to your computer and detected by your device management software.

Requirements:

On Windows Vista and 7, the latest version of Windows Mobile Center is required.

Previous Windows versions (Windows 2000 and above) require the installation of the Microsoft ActiveSync.

ATTENTION:

Make sure your device is supported by Windows Mobile Center, or that you're using the appropriate version of Microsoft ActiveSync before using Litebase Conduit.

Usage:

There are the following folders on device and on desktop:

On device:

\Litebase_DBs

On desktop:

c:\Litebase_DBs\from
c:\Litebase_DBs\to

These folders will be created on the first time you run Litebase Conduit. The “from” and “to” folders are relative to device, so:

1. “from” means “from pda to desktop”. The files from the folder (\Litebase_DBs) on the PDA will be copied to the folder (c:\Litebase_DBs\from) on the desktop;
2. “to” means “to pda from desktop”. The files from the folder (c:/Litebase_DBs/to) on the desktop will be copied to the folder (\Litebase_DBs) on the PDA;

Litebase Conduit is available with the LitebaseSDK on the path:

(LitebaseSDK installation folder)\dist\samples\LitebaseConduit\install\win32

(The default path is C:\LitebaseSDK\dist\samples\LitebaseConduit\install\win32)

ATTENTION:

Litebase Conduit will only work on PALM OS devices if the file LitebaseConduit.prc is installed on the PDA. Litebase installer for PALM OS already includes this file.

Litebase Conduit is configured using the Conduit command line options:

- /rp – Registers the conduit for PALM OS devices.
- /rw – Registers the conduit for WinCE and Windows Mobile devices.
- /ra – Registers the conduit for PALM OS, WinCE and Windows Mobile devices. (It's the same as running the conduit twice with the options /rp and /rw).
- /sp – Synchronizes the files from a PALM OS device.
- /sw – Synchronizes the files from a WinCE or Windows Mobile device.
- /cp, /cw or /ca – Shows Litebase Conduit configuration window. This configuration is shared to all registered platforms, so it doesn't matter which of the three options is used.
- /up – Unregisters the conduit for PALM OS devices.
- /uw – Unregisters the conduit for WinCE and Windows Mobile devices.
- /ua – Unregisters the conduit for PALM OS, WinCE and Windows Mobile devices. (It's the same as running the conduit twice with the options /up and /uw).

Although you can use the /sp or the /sw options to force a synchronization, this is not necessary. The Litebase files will be automatically synchronized whenever a PALM OS device performs a HotSync operation, or a WinCE/Windows Mobile device is connected to your computer and detected by your device management software.

If you have HotSync installed, you can open the Litebase Conduit configuration window through the *Customize* menu option (*Personalizar* in portuguese). Look for the `LitebaseConduit` entry and double-click it, or click on the button labeled *Change* (*Mudar* in portuguese).

On TotalCross, `Conduit` is an abstract class which extends `totalcross.ui.MainWindow`. A conduit application must extend this class and implement the methods protected abstract void `doConfig()` and protected abstract void

doSync(). These methods are used to handle the two states of a conduit: configuration and synchronization. In the Palm Desktop Conduits, the configuration occurs when the user selects the *customize* option from the HotSync menu icon. The synchronization happens when the user presses the hotsync button on the cradle or on the PDA.

The user interface creation must be handled inside the two conduit methods. A good way to separate the synchronization's logic from the configuration's is to create two classes that extend `totalcross.ui.container` and implement the logic inside each class. In the methods `doConfig()` and `doSync()` a swap to these classes must be done.

```
public class LitebaseConduit extends Conduit
{
   static
   {
      totalcross.sys.Settings.applicationId = "LBcn";
   }

   public LitebaseConduit()
   {
      super("LitebaseConduit", "LBcn", "TotalCross/AllTests",
                                                          TAB_ONLY_BORDER);
   }
   protected void doConfig()
   {
      setTitle("Litebase Conduit - Configuration");
      swap(new SetupPanel());
   }
   protected void doSync()
   {
      setTitle("Litebase Conduit - Synchronization");
      SyncPanel sp = new SyncPanel();
      sp.targetAppPath = this.targetAppPath;
      swap(sp);
   }
}
```

The application and the conduit must have the same `appCreatorId`, which should be used in the creation and registration of the conduit.

PART II – APPENDIXES

APPENDIX I – COPYRIGHT

All the contents of this tutorial, including text, programs, applets, source code, and images are copyrighted and are owned by SuperWaba Ltda, all rights reserved. No material can be reproduced and/or distributed electronically, in print or otherwise without the express written permission of SuperWaba Ltda.

The use of the source code examples is only permitted by the company that purchased this subscription.

APPENDIX II – NEWS

All the companion was updated. Please take an overall look at it.