elondra

OpenBaseMovil®

openbasemovil-db Developer Guide

http://www.openbasemovil.org

Version: 1.0

Author: Narciso Cerezo

Date: 2008-01-10

bt Innovative Technological Company, 2007
ANCES and the Spanish Industry Ministry

Index

1. Introduction

This document provides an overall description of the capabilities of the OpenBaseMovil-db J2ME database engine along with the common usage of the current JDBC-like interface.

It also explains and defines the proposed SQL support for OpenBaseMovil-db, as requested by some of you. We think that SQL is not a must, not for a mobile platform like this targeted initially to a wide range of cellular phones. But it's clear that SQL as a well known standard and it can be very helpful for certain situations.

Along the document the different SQL statements that are supported are explained in detail with their syntax and examples.

The SQL engine will be a part of the library once it's ready, but will be optional. It will not be provided as a different library, but the build process of your application will not include it if you don't use it.

It is meant to provide basic ANSI SQL support, and you must be aware that the SQL parsing will always be slower than accessing the API directly. So if you decide to use it, once it's implemented, you must balance and think very carefully if you really need it.

2. The OpenBaseMovil-db database engine

2.1 Overall description

The OpenBaseMovil-db J2ME database engine is a relational database engine with no built-in limits in the number of databases, tables, indexes, rows per table or any other dimension. It is only limited by the capabilities of the device that runs the database and the available amount of memory (storage memory, typically internal flash or card memory).

After years of investigation and testing over different devices we discovered that each J2ME device (the initial target platform form OpenBaseMovil-db) has it's own caveats and bugs. Not just from one manufacturer or model to a different one, but even from one firmware revision to a different one of the same device.

It happens that some implementations have size limits on the underlying RecordStore system (usually from 32Kb up to 500Kb), so even if you have 5Mb of memory (for example) each RecordStore is limited to a maximum size. Other might impose a limit on the size of a record in the RecordStore, but that is less usual.

Some implementations do not allow you to open more than a few RecordStores at the same time, and some have implementation bugs that cause storage memory leaks.

It is very usual that on implementations that do not impose a hard limit over the size of a RecordStore (such as SonyEricsson and Nokia S60 devices), the performance gets much more impacted by the number of records than by the size of them.

For all these reasons we divided the engine in two layers: db and storage.

The storage library is an independent library (OpenBaseMovil-storage) and it's purpose is to deal with all that issues and limitations and create an abstraction layer for the database engine.

Currently the storage library cares of things such as opening and closing the stores in the most efficient and fail-proof way, it uses multiplexing to store more than one logical record in a single physical record an thus improve the access speed, and compression on platforms that limit the size of the RecordStores. This compression comes at a price, but is usually compensated because the RecordStore implementation itself at the Java Virtual Machine level is much simpler and that causes the RecordStore to be much faster on that limited devices (this is something that is very notable on Nokia S40 when compared to S60).

The next big features at the storage library will be name-spaces and file splitting.

Name-spaces will be like a small file system with a single level of folders, so you can have different name spaces that have files with the same name. This is specially interesting for applications with multiple databases, such as OpenMidsets.

On the other hand, the most challenging but important feature is file splitting. For those devices that have a limit on the RecordStore size (BlackBerries, almost all Motorolas, Nokia S40) it will mean the end of such limitation, as it will map a single file name to multiple RecordStores in the underlying system. In a first implementation there will be still a limit in the size of a single record, which will no be bigger than the imposed RecordStore size.

For devices with no limitation, it will also have some benefits for big databases where even the multiplexing feature does not suffice to prevent performance degradation. Don't worry, this is rare and only for really big tables of more than 5,000 physical rows (depending on the row size and thus the multiplexing value it can mean 250,000 rows).

One thing that you must take into account when designing your database are blob fields (BLOB, IMAGE and VBLOB), and to a lesser extent VARCHAR fields. The engine calculates the maximum size of a row (and thus the multiplexing factor) using the maximum length of VARCHAR fields and the size of the other types except for BLOB fields. If you use a BLOB field the engine will apply a

multiplexing factor of just 2, and if you include an IMAGE or VBLOB a simple 1 (no multiplexing).

So it might be better to denormalize a table with blob fields into two linked tables, so the one with no blobs will have a much better performance.

Example:

```
Suppose you have table for storing to-do items (a task list) with a blob field for an
optional voice recording.
The traditional approach will be to issue a SQL statement like this one:

CREATE TABLE todo (
    item_date TIMESTAMP NOT NULL,
    description VARCHAR(100) NOT NULL,
    voice_note VBLOB
)

This table will be assigned a multiplexing factor of 1 because of the VBLOB field, so it
will suffer when you have a lot of rows (above 3000 or so depending on multiple factors),
even if each row will not have a voice recording and thus will only have maximum size of
212 bytes (VARCHAR fields are encoded with UTF-8 and thus have a maximum size of 2n+1
bytes, plus 2 bytes of headers per field).

If you instead create the tables like this:

CREATE TABLE todo (
    item_date TIMESTAMP NOT NULL,
    description VARCHAR(100) NOT NULL,
)

CREATE TABLE todo_notes (
    todo_id LONG NOT NULL FOREIGN KEY REFERENCES todo,
    voice_note VBLOB
)

The first table will have a multiplexing factor (as of version 3.0) of 38, thus a single
physical record will hold up to 38 logical records. And that will improve the speed of the
table a lot. The only penalization is that you will waste 9 bytes more for each voice
recording, but that is a small penalty for the achieved speed.
```

2.2 Storage

2.2.1 Database

OpenBaseMovil-db represents a database as a Store in the storage system that contains both the definition and the status of the tables in the database.

Currently this Store has the name of the database plus “_db” as its name.

In the next release of the storage engine with name-spaces, each database will have a name-space of it's own, and the files will be renamed to a standard name across databases.

The commercial version (BaseMovil-db) uses also a change index for faster change synchronization, named after the database name plus “_mix”. You better not use that name so your application can be compatible with the commercial version of the database.

Example

```
This SQL statement:

CREATE DATABASE mydb;

will create a store named "mydb_db" and in the commercial version also a store named
"mydb_mix".
```

2.2.2 Tables

OpenBaseMovil-db represents each table as a Store in the underlying storage system represented by the OpenBaseMovil-storage library. Currently, all databases use the same name space in the storage system, but in future releases of the storage system the concept of name space will be introduced so each table is a Store in the name space defined for the parent database.

This means that for the current version, no two tables may have the same name even if they belong to different databases.

All tables have always a field called “remote_id” that is the primary key for the table. It is an auto-decrement value of data type LONG.

This primary key is like an auto-increment column, but it starts with -1 and goes down. This is needed to keep the compatibility with the synchronization engine of BaseMovil, that might be open sourced in the future.

The device uses negative values that are later substituted by positive values when they're synchronized and consolidated by the sync server.

Keep in mind one important thing about tables: When you delete a row it is not really deleted, just marked as so. This is needed for the synchronization process. The commercial version of the database performs the actual deletion of the row once the server acknowledges the operation, but in this open source version you must manually do it by calling the “pack” method on the table or database.

2.2.3 Indexes

OpenBaseMovil-db (for J2ME) uses a mobile optimized implementation of BTREE indexes.

Every table has always an implicit index created over the synthetic primary key (“remote_id”), which is named in the storage system using the pattern `si_tablename_id`.

Indexes can be created over VARCHAR, INTEGER, LONG, SHORT, REAL, TIMESTAMP and BOOLEAN types.

There are three types of indexes: LONG, TEXT and FULLTEXT.

LONG indexes have keys that are represented by LONG values, all but the VARCHAR type are converted to LONG values internally when the index has just one column. LONG indexes allow to search for exact values. CASE SENSITIVE has no meaning for LONG indexes.

If the index has two or more columns or a single VARCHAR column, the index is by default a TEXT index. TEXT indexes allow to search for exact values or by “starts with” values. By default TEXT indexes are case insensitive, unless CASE SENSITIVE is specified.

When two or more columns are used for a TEXT index, the values are stored internally as a concatenation of the column values using the plus sign (+) as the separator between them. Null values are translated to the “null” text.

Both LONG and TEXT indexes can be defined on nullable columns, they support the null value and you can search for it.

FULLTEXT indexes take the value of the field or fields that compose the index as a text, concatenating the text with spaces if needed, and then take the words in that text to create a word index. Punctuation signs, white space and void words are dismissed and not indexed.

For any index you can always specify the index order using the ORDER option. By default the order is 50 for LONG indexes and 30 for TEXT and FULL TEXT indexes.

The order of a BTREE index is the number that defines how many keys are stored in each tree node with the simple formula 2n+1. For example, for an order 50 index this means that each tree node can contain up to 101 keys.

Index order allow for fine-tuning of your database. The bigger the order, the lesser number of

nodes are in the tree and usually the search process is faster. But that also means that nodes are bigger, and occupy more memory, and that is an issue in a constrained environment like J2ME.

Keep in mind also that as of OpenBaseMovil-db 3.0 nodes contain both the keys and the list of matching records, so this also accounts for the node size. For example, the implicit index has always one record per key, and keys are of LONG type so they can have a big order and the node size will not be too big. TEXT indexes will also typically match a single record, but might have large keys. And FULLTEXT indexes will usually have moderate sized keys (just a single text word) but might have a big number of matching records.

Changing the index order means to drop and recreate the index, and that can take some time.

OpenBaseMovil-db only uses the default enumeration system of the J2ME RecordStore system when for some reason the implicit index is damaged and can not be used. That has a dramatic impact on the speed of iteration over the full table contents to recreate an index, but nevertheless it can be a lengthy operation depending on the index type and the table size.

2.2.4 Basic API reference

Although the API is very wide and there are many classes implied, the 99% of the time you will be dealing with the main classes and methods explained in this section.

There are four main classes: Database, Table, RowSet and Row. And there's also another one that is not usually accessed directly but under certain situations it can be very handy: Index.

2.2.4.1 Database

The Database class is the one you will use to access all the rest of classes.

A Database contains a set of Tables, each Table in turn contains a set of fields that define the structure of it and rows of actual data.

You can create and drop databases, add and drop tables from it, lookup tables to perform operations on them, and also perform maintenance operations.

Note:

There's an alternative way to create databases other than using the API. You can use an XML file with the database definition and compile to the binary form that the database engine uses at application build time using a command line tool or ant task. Then you can download or bundle that file with your application and copy it to the appropriate Store.

This process is also used for example for views and language resources thanks to the OpenBaseMovil-util library, which are defined using XML in the first case and properties files in the second, and then compiled to an Index.

The OpenBaseMovil-core library provides a pluggable interface for language resource providers and comes with a default one that uses a properties file which gets loaded into memory. This provider has a really fast access but it can use a lot of memory if your language resources are big. So the OpenBaseMovil-util library provides a language resource provider that uses an Index as the backend for resources. This has the disadvantage of being not so fast (but fast enough) and much more memory efficient.

This resources, both views and language resources, can be bundled with the application jar or downloaded, and in any case they are copied then to a Store so they can be used.

Create a database

The following line of code will create a database called “mydb”.

```
Database db = Database.create( "mydb" );
```

Connecting to an existing database

Once you have created a database you can connect to it in order to use it. The Database class keeps track of the database objects, so no matter where you call the connect method from, you will always get the reference to the same object.

It is important that you start the database before you start using it. This is typically done at application startup time, providing some information to the user. During the database start process the database reads from the storage the database structure and status and prepares the in-memory structures needed for a fast operation.

So, when your application starts you should do the following:

```
Database db = Database.connect( "mydb" );
db.start();
```

All the database operations dispatch progress events during all might-be-lengthy operations such as starting the database, adding, removing or searching for rows, and so on.

Unless you perform operations from a backround task (see OpenBaseMovil-core developer guide), those events will reach any registered listener. Usually you will register progress view as a listener so it will show up if not already displayed automatically.

Afterwards, to access the database you just need to do:

```
Database db = Database.connect( "mydb" );
```

Drop a database

Following the previous piece of example code, we can drop the database (with all the tables and indexes that conform it) with this sentence:

```
db.drop();
```

Add a table

Let's create a new table on our “mydb” database called “mytable”, with some fields and indexes.

```
Table mytable = new Table( "mytable" );
// Add a non-null string column of up to 20 characters named "name"
mytable.addColumn( "name", Constants.FT_STRING, 20, false );
// Add a nullable integer column named "counter"
mytable.addColumn( "counter", Constants.FT_INT, true );

// Now add a full-text case-insensitive index named "ix_mytable_name" on the "name" column
mytable.createFullTextIndex( "ix_mytable_name", "name", false );
// And a simple long index over the counter field
mytable.createIndex( "ix_mytable_counter", "counter" );

// Now add the table to the database, it will not be created until we perform this step
db.createTable( mytable );
```

Lookup a table

This one is really simple, and you will use it frequently as it is the way to access and create content.

```
Table mytable = db.getTable( "mytable" );
```

Other operations

There are other less frequent operations that can be performed on a database, but that are very handy sometimes. The most useful two are:

Full database packing

This will pack all the tables in the database, effectively removing the rows marked as deleted.

```
db.pack();
```

Index repairing

Errors happens, some times the storage memory gets full, the battery drains in the middle of an update operation or things like that. While it is really difficult that rows get corrupted due to the simple and fast operation they need, it is more probable that an index might get corrupted in such circumstances.

You can detect if there's something wrong and repair it with the following code snippet:

```
db.sanityCheck();
if( db.isDamaged() )
{
    db.repair();
}
```

The repair method could be invoked without the check also, it will check all tables and indexes for damages and repair the damaged indexes.

2.2.4.2 Table

A table has an structure of fields, which can reference other tables (foreign keys), and can have indexes on any of those fields.

The table has a Store named after the table which contains the actual table rows of data, and each index has it's own Store.

The table controls it's indexes and updates them automatically whenever you perform any operation (add, update or delete rows). It also drops the indexes if you drop the table.

These are the main operations on tables:

Create a new row

```
Row row = mytable.createRow();
// Set some values in the row
// ...
row.save(); // This is a shortcut to mytable.save( row );
```

Find rows using an exact match

You need an index defined over a column or group of columns to be able to find rows that match a value. You use the find method for that, but the ways and results vary depending on the index type and definition.

We will use the “mytable” of previous examples.

Single column indexes

```
RowSet rowSet = mytable.find( "name", "Smith" );
```

If the index were a TEXT index (not a FULL_TEXT index) this find operation will match rows with the exact value "Smith" in the "name" field. Depending on whether the index is case sensitive or not it will match "Smith" and/or "smith".

```
RowSet rowSet = mytable.find( "counter", new Integer( 10 ) );
```

This will find rows whose “counter” field has the exact value 10.

The find method accepts an Object as the second argument, so you can search also for booleans, longs, shorts and dates in the same way.

Multiple column indexes

If you define an index over two columns the indexed value is built concatenating the field values as strings in the order they are defined. If a value is NULL it is translated into the indexed value as “null” (without the quotes).

Let's create first a multiple column index on our table:

```
mytable.createIndex( "ix_mytable_namecounter", new String[]{ "name", "counter" } );
```

Now let's find something:

```
RowSet rowSet = mytable.find( "name+counter", "Smith+10" );
```

This will return all records that contain exactly “Smith” and “10” on the fields “name” and “counter” respectively.

Finding rows using a partial match

The current implementation of indexes allows to find not just exact values but values that start with a certain string. Accordingly, the index type must be of type TEXT or FULL_TEXT (an index on more than one column is always a TEXT index).

For most situations, using a partial match over a FULL_TEXT index is enough

You can search for more than one “start-tokens” at the same time, and by default it will match only those rows that contain words starting with all the tokens.

```
// Find rows that contain a word starting with "sm"
RowSet rowSet = mytable.find( "name", "sm" );
// This will match "Jane Smith" and "Smeagol" but not "Chasm".

rowSet = mytable.find( "name", "ja sm" );
// This will only match "Jane Smith" from the previous example cases
```

Finding all rows

If you simply want to retrieve all the rows in a table, you would write:

```
RowSet rowSet = mytable.findAll();
```

Drop the table

Simple enough:

```
mytable.drop();
```

Would drop the table along with all it's indexes.

Improving operations with open and close

The Store system uses (like the underlying RecordStore) a nested open/close schema, that means that you can call open multiple times and only the first one will really open the underlying store, but you have to call close the same number of times to actually close it.

The Store system provides shortcuts to close a Store no matter how many times open has been called.

The database engine takes good care of opening and closing the Stores in the best and most efficient way, but there are times when you can benefit of opening and closing the Store yourself.

For example, if you insert a new row, the engine opens and closes it's Store and so does the dependent Indexes, and they do it so all the operations are performed without closing the Store but when insert finishes all are closed. This is important, as some devices do not allow you to open too many Stores simultaneously.

But if you are about to insert, lets say 10 rows one after another. That would mean to open and close the Stores 10 times, and that is somehow costly.

It is much faster if you previously open and then close the stores:

```
// The open/close methods open and close the Store associated with the Table data
// But openTree/closeTree open and close both the Store with Table data and the Stores for
// the dependent indexes
try
{
    mytable.openTree(); // Open all dependent stores

    // Insert the 10 rows
}
finally
{
    mytable.closeTree(); // Make sure to close the stores
}
```

2.2.4.3 RowSet

A RowSet is a collection of Rows that can be traversed, sorted and filtered. It is very similar to a JDBC ResultSet, but it always allows you some operations that are not always available with ResultSets such as moving to a concrete position or traversing the set up and down as many times as needed.

Typical use of a RowSet

Lets take one of the previous example as an starting point:

```
try
{
    mytable.open();
    RowSet rowSet = mytable.findFuzzy( "name", "sm" );
    while( rowSet.next() )
    {
         System.out.prinlnt( "Name:" + rowSet.getCurrent().getString( "name" ) );
```

```
    }
}
finally
{
    mytable.close();
}
```

When you issue the find or findFuzzy command, the RowSet is populated with the *pointers* to the actual records that match the criteria, but the records themselves are not retrieved. That is why we call open and close before iterating the result, as that implies opening the Store just once and not as many times as records.

As you can see, it behaves just like a ResultSet. Initially the RowSet is positioned before the first element.

Moving to a concrete record

You can navigate through the RowSet forward, backward or moving to a concrete position, like before the first, after the last or an exact position.

```
// move forward, returns true if the cursor is over a row, false if no more rows are left
// and so it gets positioned after the last row
rowSet.next()

// move backward, just the opposite
rowSet.previous();

// move to before the first record
rowSet.beforeFirst();

// move to after the last record
rowSet.afterLast();

// move to the 10th record, positions start at 0 like everything in Java and unlike JDBC :-)
rowSet.goTo( 9 );
```

Sorting data

RowSets allow you to sort data ascending or descending over a field of the RowSet. It's done with a QuickSort but nonetheless it means actually fetching every row in the RowSet so it can take some time depending on the RowSet size.

```
rowSet.sort( "name" ); // Sort ascending
rowSet.sort( "name", RowSet.DESCENDING ); // Sort descending
```

Filtering data

And also you can filter the data in the RowSet to meet a certain criteria. This kind of filtering relies on a Filter class, and it traverses all the RowSet, fetches all the Rows and calls the filter to know if the row meets the criteria or not.

After you have filtered a RowSet you can use it in the normal way, but it will navigate only through the matching rows if any.

Then you can clear the filter or you can pack it. Clearing means that again all the rows in the initial RowSet are visible. Packing means that the filter is cleared but only the rows that matched the applied filter are now in the RowSet, so if you apply a new filter you will be refining the previous one and not the initial result.

```
rowSet.applyFilter(
    new MyFilter implements RowFilter() {
```

```
        public boolean matches( Row row )
        {
             // match only rows with a counter less than 10
             return row.getInt( "counter" ) < 10;
        }
    }
);

// To clear the filter and return to the initial result
rowSet.clearFilter();

// Now filter and get only those with a counter greater than 10
rowSet.applyFilter(
    new MyFilter implements RowFilter() {
        public boolean matches( Row row )
        {
             return row.getInt( "counter" ) > 10;
        }
    }
);

// If we pack the RowSet we will have a non-filtered RowSet but that contains only the
// rows that have a counter greater than 10
rowSet.packFilter();

// If we now filter out those with a counter less than 20 we end up with those between 10
// and 20
rowSet.applyFilter(
    new MyFilter implements RowFilter() {
        public boolean matches( Row row )
        {
             return row.getInt( "counter" ) < 20;
        }
    }
);

// The example is just illustrative, but a bit silly, as we could have achieved the same
// with just one filter
```

2.2.4.4 Row

The Row represents a single logical row in a Table, usually accessed through a RowSet.

The Row has setters and getters for all the supported data types, and that is the way you can get and set values for it.

Getter and setters provide conversion between data types when possible.

Rows also have shortcuts to remove the row and to save it (save will create or update the row).

Example

```
Row row1 = mytable.createRow();
row1.setField( "name", "Milo" );
row1.setField( "counter", new Integer( 10 ) );
row1.setField( "counter", "10" ); // This also works
row1.save(); // Creates the row

int counter = row1.getInt( "counter" ); // Get the counter field as an int
Integer counter1 = row1.getField( "counter" ); // getField returns the native object
String counter = row1.getString( "counter" ); // Get the counter field as a String
```

2.2.4.5 Index

It is less usual that you would use an Index directly, but you can always do.

Indexes can store Table references or any other kind of Serializable object (see OpenBaseMovil-core developer guide) as the data element of the pair key-data that you insert in the index.

You can use an Index alone to use it like a persistent but fast Hashtable.

Example

```
// Create the standalone index named "ix_myindex", with order 30, string keys and case
// insensitive
Index index = new Index( "ix_myindex", 30, Index.KT_STRING, false, false );

// now insert some values
index.insert( "key0", "hello!" );
index.insert( "key1", "good bye!" );

// Find them
String hello = (String) index.find( "key0" );

// Remove values
index.remove( "key1" );

// And if you want, you can drop it also
index.drop();
```

3. SQL Engine Syntax

3.1 Notation

The following sections use this notation for defining the statement syntax:

Words in plain upper case are keywords: `CREATE`

Words in lower case and italics are identifiers or string/numeric literals, names for elements like tables, columns or indexes: *`mydatabase`*

Words in underscored lower case reflect more complex grammatical constructions that are defined afterwards: `column_definition`

Words in bold lower case declare the definition of a previously referenced grammatical construction, and are followed by a colon: **`column_definition:`**

Parenthesis are literal characters.

Optional constructions are enclosed in square brackets `[ ]`.

Mutually exclusive options are separated by pipe characters `|`.

String must be single quoted.

List of elements are denoted by a comma followed by three dots `,...`

3.2 Data Definition Statements

3.2.1 CREATE DATABASE

```
CREATE DATABASE db_name
```

CREATE DATABASE creates a database with the given name.

An error occurs if the database exists.

Rules for allowable database names are given in Section 4.1, “Database, Table, Index and Column Names”.

Example

```
CREATE DATABASE mydb
```

3.2.2 CREATE TABLE

```
CREATE TABLE tbl_name (
    create_definition [,...]
)

create_definition:
column_definition
| INDEX index_name [CASE SENSITIVE] [ORDER(order)] (index_col_name,...)
| FULLTEXT [CASE SENSITIVE] (index_col_name,...)

column_definition:
col_name data_type [NOT NULL| NULL] reference_definition

data_type:
VARCHAR(length) | INTEGER | LONG | SHORT | REAL | TIMESTAMP | BOOLEAN | BLOB | IMAGE |
```

```
LONGBLOB

reference_definition:
REFERENCES tbl_name
[ON DELETE reference_option]

reference_option:
RESTRICT | CASCADE | SET NULL | NO ACTION
```

CREATE TABLE creates a table with the given name.

Rules for allowable table names are given in Section 4.1, “Database, Table, Index and Column Names”.

The table is created in the current database (see USE statement). An error occurs if the table exists or if there is no current database.

If neither NULL nor NOT NULL is specified, the column is treated as though NULL had been specified.

Example

```
CREATE TABLE mytable (
    name VARCHAR(10) NOT NULL,
    description VARCHAR(50),
    counter INTEGER,
    user1 VARCHAR(100),
    ref LONG REFERENCES lookuptable ON DELETE CASCADE,
    INDEX ix_counter ORDER 80 ON ( counter )
)
```

3.2.2.1 Foreign Keys

As all tables have always a single synthetic LONG primary key, all columns that are a foreign key to another table must be of type LONG, and only the table name is needed.

As of OpenBaseMovil-db 3.0 the definition of reference options for ON DELETE action is supported, though not implemented.

3.2.3 CREATE INDEX

```
CREATE INDEX [CASE SENSITIVE] [ORDER(order)] index_name ON table_name (index_col_name,...)
```

Or:

```
CREATE FULLTEXT [CASE SENSITIVE] index_name ON table_name (index_col_name,...)
```

CREATE INDEX enables you to add indexes to existing tables.

You normally create indexes at table creation time with the CREATE TABLE statement. See the Section 2.2.3 “Indexes” for more information.

Example: create an index on a single column

```
CREATE INDEX ix_sample1 ON mytable ( name )
```

Example: create a full text index on two columns

```
CREATE FULLTEXT ix_sample2 ON mytable ( name, description )
```

3.2.4 DROP DATABASE

```
DROP DATABASE db_name
```

DROP DATABASE will drop a full database, with all it's tables and indexes.

3.2.5 DROP TABLE

```
DROP TABLE table_name
```

DROP TABLE drops a table from the database, along with all it's indexes.

3.2.6 DROP INDEX

```
DROP INDEX index_name
```

DROP INDEX removes an index on a table.

3.2.7 ALTER TABLE

```
ALTER [WITH PACK] TABLE table_name
alter_specification [,...]

alter_specification:
ADD COLUMN column_definition
| ADD FOREIGN KEY reference_definition ON column_name
| DROP COLUMN column_name
| DROP FOREIGN KEY column_name

column_definition:
col_name data_type reference_definition

data_type:
VARCHAR(length) | INTEGER | LONG | SHORT | REAL | TIMESTAMP | BOOLEAN | BLOB | IMAGE |
LONGBLOB

reference_definition:
REFERENCES tbl_name
[ON DELETE reference_option]

reference_option:
RESTRICT | CASCADE | SET NULL | NO ACTION
```

ALTER TABLE enables you to change the structure of an existing table. For example, you can add or delete columns, and add or delete foreign keys.

The syntax for many of the allowable alterations is similar to clauses of the CREATE TABLE statement.

See Section 3.2.2, “CREATE TABLE”, for more information.

Adding columns to an existing table has the restriction that the column mus be nullable, and it is a fast operation since it just modifies the table structure but no data.

Dropping columns can be far more slow, depending on whether you specify the WITH PACK option or not.

If you specify the WITH PACK option, the table is fully traversed and the fields are removed from each row. If you don't, the data is left on the table but ignored when reading the rows from the storage system. The columns will be physically removed only when you update a row.

3.3 Data Manipulation Statements

3.3.1 DELETE

```
DELETE FROM table_name
[WHERE where_condition]
```

The DELETE statement deletes rows from table_name and returns the number of rows deleted. The WHERE clause, if given, specifies the conditions that identify which rows to delete. With no WHERE clause, all rows are deleted.

where_condition is an expression that evaluates to true for each row to be deleted. It is specified as described in Section 3.3.3, "SELECT".

As stated, a DELETE statement with no WHERE clause deletes all rows. It is the same as the TRUNCATE TABLE statement. See Section 3.3.4, "TRUNCATE".

3.3.2 INSERT

```
INSERT INTO table_name ( column_name, ... ) VALUES ( expression, ... )

expression:
NULL | string_value | numeric_value | function
```

Or:

```
INSERT INTO table_name ( column_name, ... ) SELECT ...
```

The INSERT statement allows you to insert new rows into an existing table.
The first format specifies the name of the columns whose values you are about to specify in the VALUES part of the statement.
An expression can be the NULL keyword, a string value, a numeric value or the return value of a function.
Type conversion may happen on specified values.
As of this version only built-in functions can be used. See Section 3.5 "Built-in functions".

The second form will take the values from a SELECT statement. The values of the result set of the SELECT statement must match the fields to be inserted, otherwise an error is thrown.

3.3.3 SELECT

```
SELECT select_expr, ...
FROM table_references
[WHERE where_condition]
[ORDER BY col_name [ASC | DESC]]

select_expr:
* | list

list:
NULL | string_value | numeric_value | function | [table_name_or_alias.]column_name

table_references:
table_name [table_alias] [JOIN table_name [table_alias] WITH table_name_or_alias,...]

where_condition:
column_name operator condition_expression
```

```
| column_name STARTING (string_value[,...])
[ AND where_condition | OR where_condition ]

operator:
= | < | > | <= | >= | <>

condition_expression:
NULL | string_value | numeric_value
```

SELECT statement allows you to retrieve values from one or more tables optionally expressing a condition that selected rows must meet, and with an optional order.

You can select from a single table or from any number of tables, provided that the joined tables have a field with a foreign key to the master table.

If you use an equality operator and there is an index on the column the index will be used, if not, a filter will be applied (it is much slower).

3.3.4 TRUNCATE

```
TRUNCATE TABLE table_name
```

The TRUNCATE TABLE statement removes all the data from a given table and indexes, and resets the primary key to the initial value.

3.3.5 UPDATE

```
UPDATE table_name
SET col_name1 = expression [,... ]
[WHERE where_condition]

where_condition:
column_name operator condition_expression
| column_name STARTING (string_value,...)
[ AND where_condition | OR where_condition ]

expression:
NULL | string_value | numeric_value | function

operator:
= | < | > | <= | >= | <>

condition_expression:
NULL | string_value | numeric_value
```

The UPDATE statement updates the contents of one or more rows in a table.

The WHERE clause is equivalent to the one in the SELECT statement.

3.4 Additional statements

3.4.1 USE

```
USE database_name
```

Makes a database the current database.

3.4.2 PACK TABLE

```
PACK TABLE table_name
```

The pack operation removes all rows marked as deleted.

3.4.3 PACK TABLE

```
PACK DATABASE
```

Packs all tables in the current database.

3.5 Built-in functions

The SQL engine provides some built-in functions.

3.5.1 COUNT(*)

When used as the single expression in a SELECT statement, return a RowSet with a single row and a single field of INTEGER type with the number of matching rows.

3.5.2 NOW()

Returns the current time stamp.

4. Names and Data types

4.1 Database, Table, Index and Column Names

Names can be up to 26 characters, can contain letters, numbers and underscore characters.

With the next release of the Storage library this restrictions will relax to some extent, the length will be increased for example.

4.2 Data types

The following table illustrates the data types supported and their equivalent constant in the API.

OpenBaseMovil®

http://www.openbasemovil.org

elondra

http://www.elondra.com

BaseMovil®

http://www.basemovil.com

http://developer.basemovil.com

Elondra
www.elondra.com
Plaza del Pan, 11 - 45600 Talavera de la Reina (Toledo)
Spain
Phone: +34 911.201.709