From Local to the Cloud – PART 2

Michael
McCreary

#LiveCodeGlobal

What is a database?

A structured collection of data

Relational databases

- A database schema - a collection of tables
- Database contents - rows or records in the tables
- A query language - SQL
- A database engine

Accessed from LiveCode using revDB

Database types

Local database

- Single user
- File based
- E.g. SQLite

Server database

- Multi user
- E.g. MySQL, PostgreSQL, Oracle

Creating our SQLite database in LiveCode

Use revDB

Choose a location for our database

Open a connection to our database - `revOpenDatabase`

Use the unique ID for that connection to perform database operations

Store database schema in custom property

Set up database schema on first run

#LiveCodeGlobal

Creating

```
CREATE TABLE notes

(

  id INTEGER PRIMARY KEY AUTOINCREMENT,

  title TEXT NOT NULL,

  body TEXT NOT NULL,

  created_date DATETIME NOT NULL,

  edited_date DATETIME NOT NULL

);
```

#LiveCodeGlobal

Inserting

```
INSERT INTO
   notes (title, body, created_date, edited_date)
VALUES
   (‘Note title’, ‘Note body’, datetime('now'),
   datetime('now'))
```

Querying

```
SELECT * FROM notes
```

Updating

```
UPDATE

   notes

SET

   title = ‘new title’, body = ‘new body’

WHERE

   id = 123
```

Unicode text and databases

The encoding of text is not fixed in LiveCode

Use `textEncode` when inserting

Before inserting, execute:

```
PRAGMA encoding='UTF-8'
```

Tells SQLite to expect UTF-8 encoded text

Use `textDecode` when fetching

#LiveCodeGlobal

SQL Injection

What if someone enters “My note’; DELETE FROM notes”

Use placeholders to protect from SQL injection:

```
revExecuteSQL sDBConnectionID, "INSERT INTO notes
(title, body) VALUES (:1, :2)", "pTitle", "pBody"
```

Using a database library

Put common tasks into a library

Promotes code reuse

Speeds up development

Script only stack

Place in message path using `start using`

Connecting

```
command libDBConnect pType, pHost, pDBName, pUser, pPass
```

- Opens a new connection to a database
- Wraps `revOpenDatabase`
- libDB manages connection ID
- Throws exception on failure

```
command libDBCloseConnection
```

- Wraps `revCloseDatabase`
- Closes any connection libDB has open

Connecting

```
try
   libDBConnect "sqlite", "notes.db"
catch tError
   -- handle any error here
end try
```

Executing SQL

```
command libDBExecute pQuery, placeholders...
```

- Executes passed SQL statement
- Pass any placeholders as additional parameter
- Wraps `revExecuteSQL`
- Returns the result of the query or throws exception on failure

#LiveCodeGlobal

Executing SQL

```
try
   libDBExecute the uDatabaseSchema of me
catch tError
   -- handle any error here
end try
```

Inserting data

```
command libDBInsert pTable, pData
```

- Pass the name of the table to insert into
- Pass the data to insert
  - Array of key value pairs
  - Key - the name of the field
  - Value - the value we want to insert
- Builds the insert query
- Escapes and encodes values
  - Prefix key with *f for functions
  - Bypasses escaping and encoding
- Returns the ID of the newly created row

#LiveCodeGlobal

Inserting data

```
local tData
put "Note title" into tData["title"]
put "Note body" into tData["body"]
put "datetime('now')" into tData["*f created_date"]
put "datetime('now')" into tData["*f edited_date"]
try
   local tNoteID
   libDBInsert "notes", tData
   put the result into tNoteID
catch tError
   -- handle any error here
end try
```

Updating

```
command libDBUpdate pTable, pData, pID
```

- Pass the name of the table to update
- Pass the data to update as array
- Pass the ID of the row to update
- Builds the update query
- Escapes and encodes values
- Returns the number of rows updated

#LiveCodeGlobal

Updating

```
local tData

put "New note title" into tData["title"]

put "New note body" into tData["body"]

put "datetime('now')" into tData["*f edited_date"]

try

   libDBUpdate "notes", tData, "123"

catch tError

   -- handle any error here

end try
```

Fetching data

```
function libDBFetchAsText pQuery, placeholdders
```

- Wraps `revDataFromQuery`
- Pass the query to execute
- Pass any placeholders as additional parameters
- Returns the rows as CSV
- Good for fetching single fields as text
- Poor for fetching more complex data

Fetching data

```
function libDBFetchAsArray pQuery, placeholdders
```

- Wraps `revQueryDatabase`
- Pass the query to execute
- Pass any placeholders as additional parameters
- Returns the rows as an array
  - Two dimensional array
  - First dimension - integer index array of rows
  - Second dimension - row data as key, value pairs
  - Just like a datagrid's `dgData` property

#LiveCodeGlobal

Fetching data

```
set the dgData of group "notes" to libDBFetchAsArray("SELECT *
FROM notes")



local tNoteTitle

put libDBFetchAsText("SELECT title FROM notes WHERE id = :1",
"123") into tNoteTitle
```

Searching

`WHERE` keyword adds clauses to `SELECT` command.

```
SELECT * FROM notes WHERE title = 'foo'
```

Searching

```
SELECT * FROM notes WHERE title LIKE '%foo%'
```

- `LIKE` searches for matches
- `%` matches any sequence of characters

Searching

```
SELECT * FROM notes WHERE title LIKE '%foo%' OR body LIKE '%foo%'
```

Use boolean operators (`AND`, `OR`) to search on multiple fields

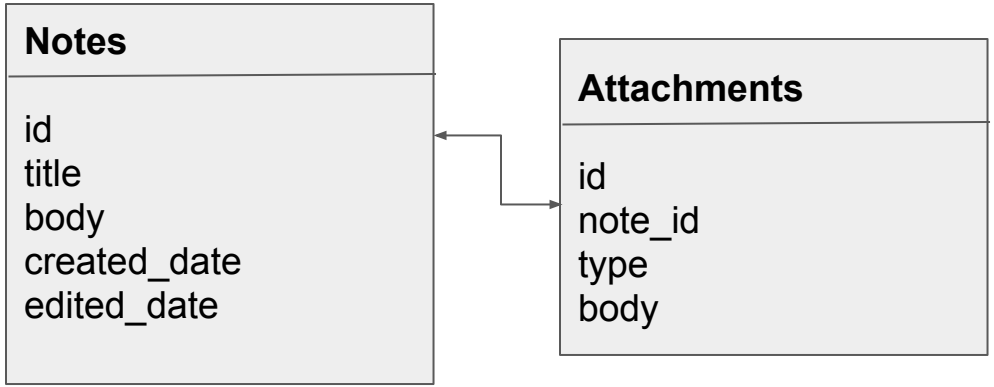
Adding attachments

- Create a new attachments table
- Allows for a note to have more than one attachment
- Map attachment to note using note ID

Adding attachments

```
CREATE TABLE attachments
(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  note_id INTEGER NOT NULL,
  type TEXT NOT NULL,
  data BLOB NOT NULL
);
```

Fetching attachments




```
SELECT * FROM attachments WHERE note_id = 1234
```

Binary data

Handle binary as a special case

- Use blob as field type
- Prefix placeholder with *b when inserting and updating

Adding categories

Add a categories table

Add a mapping table

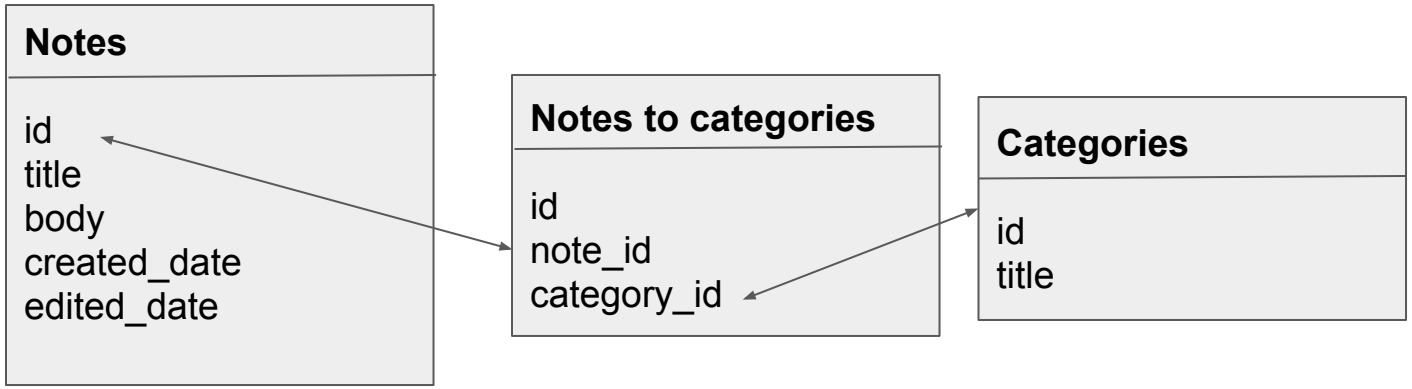
- Maps notes to categories
- Use note ID and category ID as mappings
- Allows for multiple notes to be mapped to same category

Adding categories

```
CREATE TABLE categories
(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL
);
CREATE TABLE notes_to_categories
(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  note_id INTEGER NOT NULL,
  category_id INTEGER NOT NULL
);
```

#LiveCodeGlobal

Mapping categories



Fetching note categories

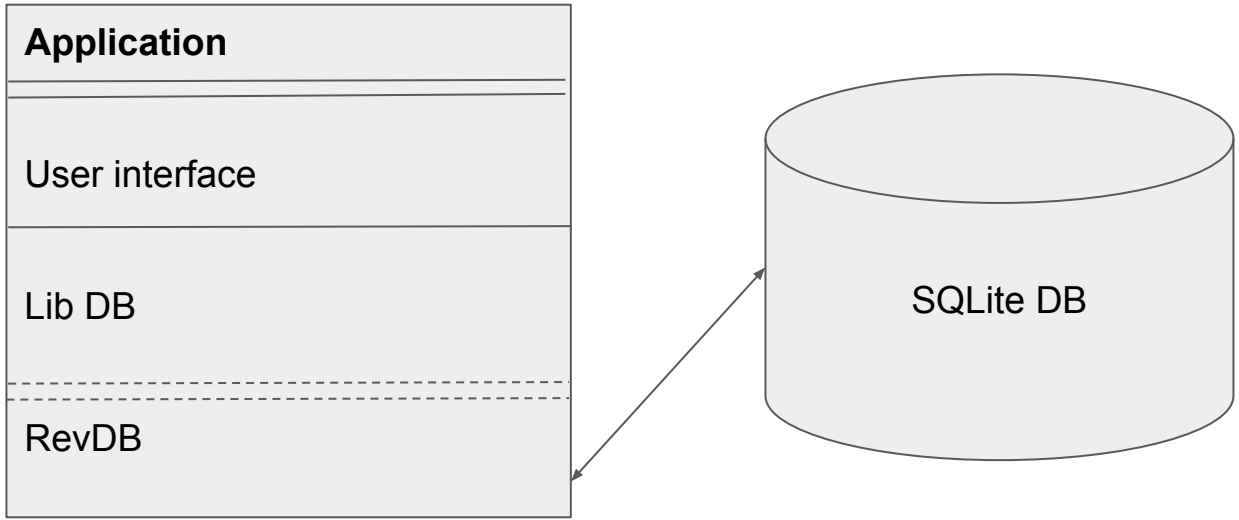
```
SELECT
   categories.title
FROM
   notes_to_categories, categories
WHERE
   notes_to_categories.category_id = categories.id
   AND notes_to_categories.note_id = 123
```

#LiveCodeGlobal

Filtering by category

```
SELECT
   notes.*
FROM
   notes, notes_to_categories, categories
WHERE
   notes.id = notes_to_categories.note_id AND
   categories.id = notes_to_categories.category_id AND
   categories.title = ‘red’
```

Application Structure



Moving to the cloud

Use a data layer

An intermediate layer between user interface and lib DB

Completely separates user interface from database

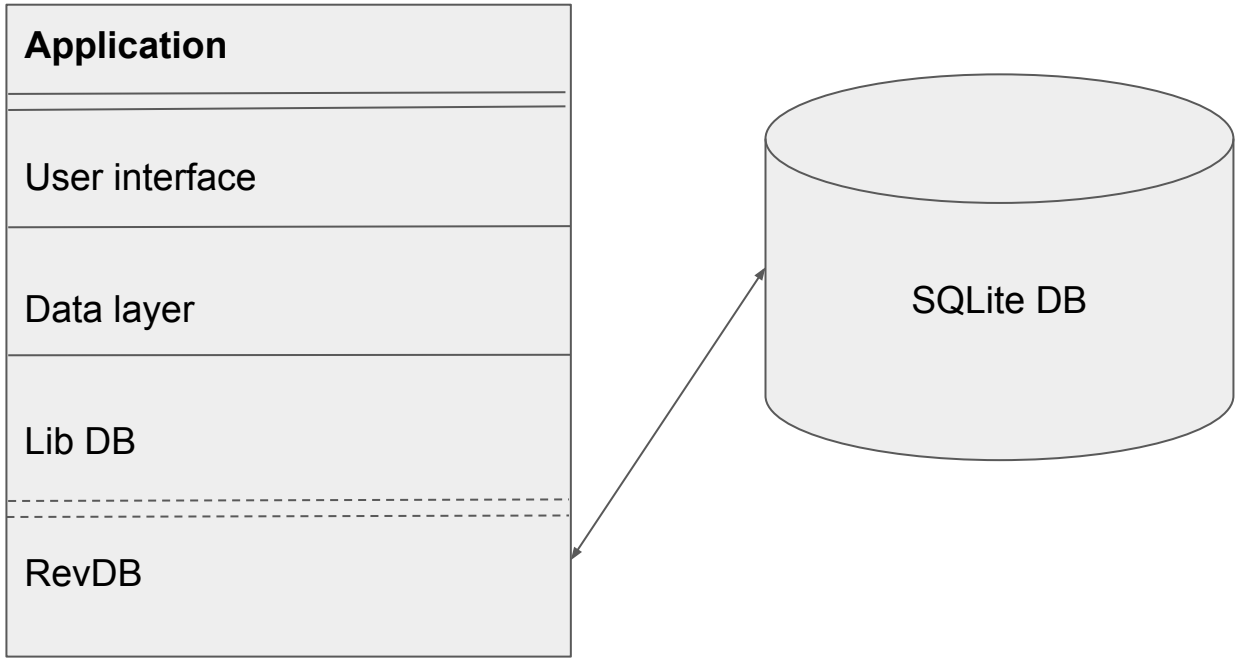
Abstracts the notion of a data store

Data store can be implemented in any way e.g.

- SQLite DB
- Text files
- Server side database

#LiveCodeGlobal

Inserting a data layer - local DB



#LiveCodeGlobal

Inserting a data layer - cloud DB