Text Data Storage Method Modification for SQLite Relational Database

I G Borovskoy¹, E A Shelmina¹, N V Zarikovskaya¹

¹ Tomsk State University of Control Systems and Radioelectronics, 40, Lenina ave., Tomsk, 634050, Russia
E-mail: igdoca@mail.ru, eashelmina@mail.ru

Abstract. The paper is devoted to the studies of the issue related to the support of national alphabets in the applications working with the SQLite relational database. The present paper represents the analysis of existing methods of modification of potential Unicode embedding into SQLite database, a database modification solution algorithm, and comparative analysis of the proposed approach versus current methods. An application working with SQLite with a modified module for national alphabet support has been developed as a result of the performed studies.

1. Introduction

Software developers often have to use databases for data storage in their projects. However, many of them try to avoid it, since they know the amount of additional expenses associated with the SQLite server installation. This problem turned out to be solved with SQLite appearance. No additional software is required to use SQLite. The work is carried out locally in the project; meanwhile, the whole SQL language power is used.

SQLite is an embedded relational database supplied with the source codes. It is designed to provide for the traditional opportunities of relational databases without relevant indirect costs. For the time of its operation the database has proved to be portable, easy to use, compact, efficient and reliable.

Data base embeddability means that it exists not as a process separate from the client process but is a part of some application. A casual observer will not notice that the application utilizes a relational database management system. This relieves from the necessity of network parameter settings, firewalls, no network addresses, users and their access right conflicts. Both a client and a server work in one process, which eliminates configuration problems. Everything what a software engineer needs is already compiled in his application [1, 5].

Besides, SQLite has a modular architecture reflecting unique approaches to the relational database management. Eight separate modules are grouped into three main subsystems. They split request processing into separate tasks operating like a conveyor system. Upper modules compile requests, intermediate ones perform them, and the lower modules deal with the disc and interact with the operating system.

It is also worth mentioning that SQLite uses dynamic typing. The data type of the value therein is connected with the value itself but not with the table field. The dynamic typing is backward compatible with the statistical one. So, all SQL expressions working in traditional DBMS will stay valid also in SQLite. However, while talking about typing in SQLite we should also mention the issue
of text data storage. The point is that a problem of national alphabet usage appears when working with SQLite. Currently, there is no precise solution related to national alphabet support while working with SQLite. There are only approaches to the implementation depending on the area of application and task types.

The present paper is devoted to the studies of the modern relational SQLite database, and in particular to its modification in the storage of text data.

The study is carried out from the point of view of SQLite interaction with the applications developed in C++ programming language, although the functional area of the relational database is pretty significant.

2. Text Data Storage in SQLite

The embedded databases are represented by the products from various manufacturers, however, only SQLite is supplied with open sources and does not require license fees and designed purely as an embeddable database [2, 7].

SQLite completely supports UTF–8 and UTF–16 codes. However, there is one nuance, i.e. case insensitive like and order by does not work for string fields and text fields whose symbols go beyond the ASCII table [3].

As of the moment, there is no precise solution related to the issue of national alphabet support when working with SQLite. There are only some approaches to the implementation depending on the area of application and task type:

1. To compile SQLite with required extension, with ICU library which is available at the official Unicode Consortium site. If this addition is downloaded and embedded into a ready project, ICU extension substitutes upper() and lower() functions which are in charge of the case conversion [14].

ICU library compiling with SQLite database will solve the set up task, but will provoke several disadvantages both for the user and the developer. The main one is that the library brings a lot of unnecessary codes, which adds to the library heavy weight and, therefore, deprives it of the major advantage, i.e. its lightness [4, 9].

2. To use a standard extension which is already available, i.e. FTS3 or FTS4. Extension modules FTS3 and FTS4 provide users with the opportunity to create special tables with the full-text index support. The full-text index allows users to efficiently query the database for all rows that contain one or more words (hereinafter referred to as tokens) even if the table contains numerous large documents [6]. Although FTS3 and FTS4 come fitted with the SQLite core source code, they are not enabled by default. To build SQLite with FTS functionality enabled, it is necessary to define the macro SQLITE_ENABLE_FTS3 when compiling. New applications should also define the SQLITE_ENABLE_FTS3_PARENTHESIS macro to enable the enhanced query syntax [10, 11].

3. To create one more column (normalized) in the table. The column should contain the same information which shall be searched for, but in the upper (or lower) case. Then, to make comparison with the help of the LIKE statement on the basis of this column, but the result should be brought back to the regular column. Since the same symbols can be represented by different codes, string matching becomes impossible byte by byte. Normalization algorithms solve this problem converting a text to some certain standard type. The conversion is performed through the replacement of symbols with equivalent ones with the use of tables and rules [10].

The following drawbacks were revealed when analyzing the proposed database modification solution methods:

1. Connection of ICU to the project is an advantage for the national alphabet support, however, such projects as a rule become significantly heavier from the program weight point of view, which results in slower processes. Therefore, this solution method does not suit for small projects.

2. As for extensions already available in the database – FTS3 and FTS4, all fields used for the search can be doubled and text can be stored therein in the lower case. It is not very convenient to control correspondence of the values to the initial fields with the help of the program, since it is very
time-consuming, and if there is a large scope of different information, the database memory enlarges, which is also inconvenient.

3. To create one more column (normalized) in the table. The column should contain the same information which shall be searched for, but in the upper (or lower) case. Then, to make comparison with the help of the LIKE statement on the basis of this column, but the result should be brought back to the regular column. This also adds significant weight to the database, and when a new row appears or changes, the normalized column shall be addressed and search criteria shall be changed.

The conducted studies show that, as of the moment, neither developers nor users have a more efficient and, what is more important, precise solution for national alphabet support. Everybody solves this problem depending on the working area and project complexity. The present paper studies another problem solving option which solves the following tasks:

- Support of national alphabets in applications using SQLite relational database;
- Efficient database operation;
- Lower speed of query execution.

3. String Data Type Modification Methods
Apart from the methods considered above, there are two more solution methods for SQLite national alphabet support. The first method is to redefine string functions in source sqlite3.c code and to recompile the sqlite3.dll library on the bases of the modifications. The second method is to write a module based on the algorithm of interception of string functions “on the fly”. The developed module will be included into the implemented project working with SQLite.

3.1. Text Data Storage Method Modification for SQLite through Source Code Change
The first method is to introduce changes into the SQLite source code for string functions working with Latin alphabet by default. To recompile the sqlite3.dll library based on the modifications and connect it to the project.

The disadvantage of this modification is that every year a new sqlite3 build version appears and the compiled patch will not suit to the new version since developers do not change functions in the source codes. And it is not economically feasible to adjust to new functions and to re-write the code [12].

3.2. Text Data Storage Method Modification by Interception of Functions “On the Fly”.
The interception of a standard API SQLite string function involves the modification of some address in the process memory or some code in the function body so that when calling for this API-function the control would be ceded not to this function but to the Unicode-string comparison function which replaces standard functions of operation with strings in SQLite. Working instead of the standard function this function performs planned actions replacing encoding “on the fly” without triggering the original function of the application [4,8].

The idea of the implemented algorithm is to replace a standard string comparison algorithm with Unicode-functions, which would cause symbol recoding at the moment of entering into the database. Modifications “on the fly” are implemented through the triggering of corresponding scripts written at the stage when rules of operation with strings are set up.

Functions are implemented in the sqlite.dll library and if pointers are empty, i.e. nothing is transferred to the functions, string comparison is carried out by default. Otherwise, when simplified comparison functions are performed by default, Unicode comparison functions are hooked and, so, replace the encoding. When using the callback-function system the standard way of string comparison is replaced with the one which is implemented in the project. As a result of these modifications, the code itself does not change but only the memory code changes [13].

If the base string format does not match with the format of the string which is transferred to API, then the transferred string is recoded “on the fly” into the format of the base. The point is that SQLite tries to select a method to the format, otherwise (if formats of transferred string and registered function
differ) then encoding “on the fly” is performed. The algorithm of interception of SQLite string data “on the fly” with their replacement with Unicode-functions is based on this process.

Windows-application which is a module implementing the work with SQLite text data was developed on the basis of the considered method. This module contains the method of “functions replacement by default in SQLite with Unicode functions”. Standard actions to check the correct value of a string and specific code call are performed therein in order to implement actions for particular application of the module; in case of our algorithm, it is interception of string functions with their replacement with Unicode-functions.

So, through the connection of the developed module to an application working with SQLite the problem of string case comparison and, consequently, of displaying unnecessary text encoding can be solved. Due to the fact that the connected module does not utilize a lot of space and does not capture a lot of memory the main specific feature of the data base, i.e. its light weight, is preserved. Such a method of function interception “on the fly” can be used in any environment and adjusted to various programming languages.

The advantage of the given development is that even if a new SQLite library version appears the developed module will work with any version. The given module can be also compiled in a separate library and attached to the projects requiring SQLite in full.

4. Applications for Text Data Storage Method Modification for SQLite Relational Data Base
So, on the basis of the set problems, an algorithm of the method of text data storage in SQLite as well as an application working with SQLite have been developed.

The application improving the text data storage method in SQLite has been implemented in the Microsoft Visual Studio programming environment and is available free of charge at the site http://igdoca.bplaced.net/Data_/SQLite.exe. The application requires sqlite3.dll library (http://www.sqlite.org/download.html) and represents a notebook, a sort of a link directory which helps a user to easily write, find and, foremost, visit a site which used to be of interest to him. The application allows making sure that the developed module operates properly with the information from the SQLite data base which can be added independently, changed when in use and the content can be amended. The developed module for SQLite string data adheres to the application source code providing for the proper work with data basing, search and output.

Conclusion
The national alphabet issue was analyzed within the framework of the study, the structure of the SQLite relational data base was considered in detail, as well as its parameters which shall be modified to support national alphabets. An emphasis was put on the comparative analysis of available SQLite string data modification solution algorithms. An efficient approach allowing for string data modification through the connection of the developed module to the project using SQLite was proposed on the basis of the performed analysis.

References
[1] Chunyue Bi 2009 Research and application of SQLite embedded database technology WSEAS Transactions on Computers 8(1) 83-92
[2] Iqra Jan, Junaida Shafi 2015 Sqlite- A Better Choice as an Embedded Database for Smart Phones International Journal of Allied Practice, Research and Review 2(6) 9-15
[3] Kiran Dhokale, Nandeo Bange, Shelke Pradeep, Sachin Malave 2014 Implementation of SQL Server based on SQLite Engine on Android Platform International Journal of Research in Engineering and Technology 3(4) 8-14
[4] Sunguk Lee 2012 Creating and Using Database for Android Applications International Journal of database Theory and Appli 5(2) 31-38
[5] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi 2011 Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration PVLDB 4(8) 494–505

[6] Hossein Falaki, Ratul Mahajan, and Deborah Estrin 2011 Systemens: a tool for monitoring usage in smartphone research deployments In Proceedings of the sixth international workshop on MobiArch 25–30

[7] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin 2010 Diversity in smartphone usage In Sujata Banerjee, Srinivasan Keshav, and Alec Wolman, editors, MobiSys 179–194

[8] Sangjun Jeon, Jewan Bang, KeunDuck Byun, and Sangjin Lee 2012 A recovery method of deleted record for sqlite database Personal and Ubiquitous Computing 16(6) 707–715

[9] Linying Jiang Kun Yue Liu Yang, Heming Pang 2010 The Research of Embedded Linux and SQLite Database Application in the Intelligent Monitoring System

[10] Yubin Xu Jianwei Li, Hong Guo 2012 Research on Optimization of SQLite Indexing Mechanism Based on Red-Black Tree http://www.scientific.net/AMM

[11] Xiaoquan Chen Jihong Zhang 2012 Research and Design of Embedded Wireless Meal Ordering System Based on SQLite

[12] Susan J. Shepard 2011 Embedded Databases Can Power Emerging World of Information to Go

[13] Akshat Aranya Cristian Ungureanu Shuai Hao, Nitin Agrawal 2013 Building a Delay-Tolerant Cloud for Mobile Data.

[14] Gi-Hwan Oh Bongki Moon Woon-Hak Kang, Sang-Won Lee 2013 X-FTL: Transactional FTL for SQLite Databases