Improving SQL Query Performance on Embedded Devices using Pre-Compilation

Graeme Douglas
University of British Columbia
graeme.r.doug@gmail.com

Ramon Lawrence
University of British Columbia
ramon.lawrence@ubc.ca

ABSTRACT

Embedded devices are increasingly being used for data collection and on-device data analysis for applications in environmental and infrastructure monitoring, health and wearable computing, and sensor and mobile systems. Processing data on the device rather than transmitting it over a network for analysis reduces energy consumption, network bandwidth usage, and results in more robust and longer functioning devices. A key challenge is enabling efficient data management on devices that may only have a few KBs of memory and limited code space. Previous work has demonstrated that using relational databases is possible on embedded devices with restrictions on the queries that can be processed. In this work, we eliminate one of the key barriers to using relational technology on embedded devices, which is the massive overhead involved in SQL parsing and translation that can take up to 50% of the code and memory resources on the device. Our approach allows developers to continue to use relational APIs and SQL during development which are then pre-compiled when deployed on the device. This produces all of the benefits of relational systems without the on-device overhead and limitations. Experimental results demonstrate that query pre-compiling can reduce query parse times by up to 90% and on-device execution times by up to 50%. The technique is applicable to a wide range of embedded systems and databases.

CCS Concepts

•Information systems → Database query processing; Structured Query Language; •Computer systems organization → Embedded and cyber-physical systems;

Keywords

embedded, query, SQL, database, Arduino

SAC 2016, April 04-08, 2016, Pisa, Italy
© 2016 ACM. ISBN 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851657

1. INTRODUCTION

The relational model and SQL have significant benefits as they provide a higher level of abstraction compared to developing custom data management code. This abstraction results in increased productivity, faster time to market, and lower development and maintenance costs. Embedded devices are capable of data management [4], but their limited resources is a challenge. The smallest devices including Arduinos [6] may have as little as 2KB of memory and 32KB of code space. These resource constraints challenge deploying relational technologies on embedded platforms.

One of the most significant challenges is the parsing and translation of a SQL query into an executable plan. The parsing and validation phase of a relational database system consumes a significant amount of memory and code space. Previous embedded relational database systems such as Antelope [7] and LittleD [3] handled this challenge by reducing the complexity of SQL queries supported. Still, the parsing component consumed a high amount of code space and limits the devices that these databases could be used for. Network sensor databases such as COUGAR [2] and TinyDB [5] used pre-compiled query plans that were transmitted from controlling nodes to the embedded sensor nodes. In these architectures, the embedded nodes are dependent on the controlling node and are not designed to operate in a stand-alone fashion.

Our contribution is a SQL query pre-processor for embedded relational databases that allows the flexibility of developing using SQL without the code and run-time overhead of parsing and translating SQL queries on-device. Our system allows developers to use relational development techniques such as prepared statements and dynamic SQL that are translated at compile-time to efficient embedded code as callable and compilable C functions. Experimental evaluations demonstrate that query pre-compiling drastically reduces memory usage and improves parsing efficiency by up to 90% and overall query execution time by up to 50%, as the expensive parsing process is not performed at run-time. The technique is applicable to any embedded device and relational databases such as LittleD, Antelope, and SQLite.

The organization of this paper is as follows. In Section 2 is background on data techniques for embedded systems. Section 3 provides an overview of the approach, and Section 4 contains experimental results. The paper closes with future work and conclusions.

2. BACKGROUND

Relational database systems parse Structured Query Language (SQL) into optimized execution plans. The process starts by lexically analyzing the SQL query string into a stream of tokens. This token stream is then parsed, resulting in a parse tree. This parse tree is fed into a planner to translate and optimize into an optimized plan. Finally, the query plan is made executable.

While this approach is ideal for relational database systems on workstations and servers, resource constrained devices such as smart cards and sensor nodes cannot afford the code space (often less than 128 KB), memory (between 2KB and 64KB), and energy requirements for such query translation. Databases designed for local data storage and querying on embedded devices, such as Antelope [7] and LittleD [3], parse queries on-device. The translation approach is greatly simplified and involves directly translating the query string into an executable query plan by constructing bytecode during the parsing of the token stream. While this technique allows for ad-hoc querying, the implementation complexity of the query-translation component prevents these systems from being used on some of the most resource constrained devices. Even with this simplified query parsing and execution model, for LittleD the parser takes about 21KB of code space of a total of 57KB for the entire library. PicoDBMS [1] is an embedded database for smartcards, but it was not designed to perform SQL query processing on device.

Systems are forced to make compromises on the components implemented and the level of SQL support. Antelope and LittleD support a subset of SQL. Systems such as TinyDB [5] and COUGAR [2] are distributed data systems intended to manage information over many networked sensors. These systems perform query parsing and translation off-device. A control system exists which manages queries across the network. Although this eliminates a considerable amount of code by removing the parser and optimizer, the embedded device now becomes dependent on an external device. It also adds another level of complexity when implementing data algorithms.

For mobile devices and smart phones, embedded databases include SQLite[1] and BerkeleyDB[2]. SQLite is not able to work on the most resource-constrained devices, such as Arduinos, due to its high minimal memory and code requirements (approximately 200 KB).

Database application programming interfaces (APIs) allow developers to embed SQL in their program code which is dynamically executed and results produced. SQL statements may be static (unchanging) or contain dynamic parameters (prepared statements) which change during program execution. In an embedded database, the SQL processing occurs in the same process as the application code via the embedded database library. Even for embedded systems like SQLite, there has been no prior work on eliminating the parsing and translation process between the application and the database library. While SQLite supports prepared statements, those statements have the original SQL query string manipulated in code, parsed, and then translated into executable byte code within the library.

There is previous work in pre-compilation of queries. Relational database systems will often have a pre-compiled query plan cache to allow for efficient repeated execution of queries by avoiding re-parsing/optimization. Prepared statements are designed to allow a query to be compiled once and used multiple times. Various database API languages may perform some form of static analysis on queries embedded in code. This work is unique as the goal is to completely remove the parsing/optimization component of the embedded database without compromising functionality.

3. APPROACH

Our technique allows relational SQL queries on embedded devices without requiring the database library to parse SQL at run-time. The steps are:

- The developer defines a JSON file containing the SQL queries required by their code. This file is created and processed on the development machine.
- The developer processes the JSON file using a `make` command that calls on the database engine to parse the query and produce an executable plan. The output is a C code file with functions that can be called in the main application code. Each one of these functions implements an execution plan for a SQL query.
- In the main application, the developer calls the C functions to perform SQL queries and may pass at run-time any dynamic bind variables to be used for the query.
- The database library executes the plan directly without parsing/compilation as the SQL query has already been translated into an execution plan during the development process.

The key advantage of this approach is that the database library deployed on the embedded device can be smaller and more efficient as only the execution engine is required. Parsing and translation is performed as part of the build process which saves execution time. The trade-off is that query execution is no longer dynamic based on the data properties as it would be in a traditional relational database which may adapt its execution plan depending on current data sizes and organization. Given that embedded devices typically have well-defined tasks and data sets and query processing limits, this is a reasonable trade-off for the increased performance.

A detailed example follows. In the first step, a JSON file is constructed to define the queries that will be used in the code. In Figure 1 is an example JSON file with two queries; `query1` is a static query, while `query2` contains two parameters. Queries are defined in groups, with each group getting its own compilable file (`queries.c`). Memory limitations for the query on device can be specified on a per group basis. A path to the relational metadata is also provided for use in planning/optimization. Each query group has an array of queries with names and SQL strings. The names become the C function names (and are therefore constrained by the same constraints as C function names) and the SQL string specified gets serialized into code that executes the query.

A C program executed with `make` reads the JSON file defining the queries and for each query invokes the database engine to parse, translate, and optimize the query into a query plan. This query plan consists of a tree of operators (iterators) (i.e. select, project, join). The in-memory query

[1]http://www.sqlite.org/

[2]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/

```
[{
"filepath":     "queries.c",
"relationpath": "data",
 "queries": [
  {
   "name":  "query1",
   "query": "SELECT * FROM r;"
  },
  {
   "name":  "query2",
   "query": "SELECT 3*attr0, attr1*2, attr2%13
             FROM r
             WHERE attr0 >= ?i AND 2*attr1 != ?i;"
   }]
}]
```

Figure 1: JSON File Defining SQL Queries

plan is traversed and serialized into C code that executes the required functions in the database engine library. The exact content of the C function for a query depends on the database engine library. In our experiments, the code is for the LittleD engine, but others like Antelope and SQLite are possible. Each query gets translated into one C function. Compiled/serialized queries may take optionally in the WHERE clause a set of typed placeholders (?i for integer, ?s for string) that get translated, in order of appearance in the query string, into parameters of the corresponding C function. In Figure 2 are the C function signatures produced. These functions return an iterator to traverse the tuple results, and are passed in a memory management pointer to allow for the query to have a set amount of memory to use.

```
db_op_base_t* query1(db_query_mm_t* qmm);
db_op_base_t* query2(int param1, int param2,
                             db_query_mm_t* qmm);
```

Figure 2: C Functions for End-User Code

The developer can now use these C functions in their main application code (see Figure 3). In the example, the application sets up the maximum amount of memory it will set aside for database queries, initializes the tuple structure to access each row of the result, and then iterates through the results in a loop. The application programmer uses SQL and the flexibility of a relational engine with minimal overhead while retaining full control of memory usage on the device. When deployed on the device, all C code is compiled and loaded onto the device. There are never any SQL strings in code or memory on the device. This reduces both the size of the database library (as a parser is not needed) and the amount of code space used for string constants. String constants also consume limited memory on the device, so removing them also saves vital memory resources.

4. EXPERIMENTAL RESULTS

Experiments were conducted to determine the code savings by removing the parsing component from LittleD as well as the execution time savings for pre-compiling query plans. The experimental device used was an Arduino Mega2560 with SD card and Ethernet shield with 8 KB RAM and 256 KB of code space. It has a 16MHz 8-bit AVR processor. A variety of queries were executed on a relation consisting of 100 rows stored on a SD card. The queries contain expressions and filters on one table. The data set was 100 rows as the goal is to measure parsing time differences not execution time differences. The parse time is a constant time regardless of the number of rows processed whereas the execution time varies with data size and scales linearly for the queries tested. Each query was run 10 times with the average time across all runs presented.

```
int main()
{
   // Initialize memory available to queries
   int size = 2000;
   unsigned char mem[size];
   db_query_mm_t qmm;
   init_query_mm(&qmm, mem, size);

   // Execute the query
   db_op_base_t *iter;
   iter = query2(1, 3, &qmm);

   // Initialize memory for result tuple
   db_tuple_t tuple;
   init_tuple(&tuple, iter->header->tuple_size,
               iter->header->num_attr, &qmm);
   int col1, col2, col3;

   // Iterate through the results
   while (next(iter, &tuple, &qmm))
   {
      count++;
      col1 = getintbypos(&tuple, 0, iter->header);
      col2 = getintbypos(&tuple, 1, iter->header);
      col3 = getintbypos(&tuple, 2, iter->header);

      printf("record=[%d, %d, %d]\n",
                col1, col2, col3);
   }

   printf("# of results: %d\n", count);
}
```

Figure 3: Using Functions in End-User Code

In Table 1 are results containing the query, the parse time on device, the time for setting up the pre-compiled query, the query execution time, and percentage improvements for parsing and overall query execution. The `Pre-Compile Time` column shows the time *on the device* to initialize the query. When parsing a query, this initialization time includes the time to parse and translate the query into the execution plan. For the pre-compiled queries, the only code executed on the device is to initialize the query iterator and data structures as the execution plan is already built. Note that the time to pre-compile the query on the development machine is not shown as this occurs offline during the development process and is insignificant. The parse/pre-compile times correspond to the time to run the code in the `query2()` method in Figure 3. Execution times are measured by the time to produce all the results (i.e. the while loop with the next iterator in Figure 3).

Table 1: Parsing vs. Pre-Compilation Results

Query	Parse Time (ms)	Pre-Compile Time (ms)	% Parsing Improvement	Execution Time (ms)	% Overall Improvement
`SELECT * FROM r`	16.9	10.2	40%	18.3	21%
`SELECT attr0 FROM r`	29.7	10.5	65%	28.3	34%
`SELECT attr0, attr1 FROM r`	46.4	10.3	78%	34.1	45%
`SELECT attr0, attr1, attr2, attr3 FROM r`	79.8	10.6	87%	46.1	55%
`SELECT 3*attr0, attr1*2, attr2%13, 14+attr3*2 FROM r`	87.6	11.1	87%	90.2	43%
`SELECT 3*attr0, attr1*2, attr2%13, 14+attr3*2 FROM r WHERE attr0 >= 0 AND 2*attr1 != -1`	112	11.3	90%	137.3	41%

Pre-compiling queries rather than on-device parsing results in massive performance improvements for query setup by up to 90%. For the most complicated query, pre-compiling the query is 10 times faster. There is also a significant reduction in code space usage of over 35%. Thus, there is a win both in run-time performance and code space utilization.

A second important point is that the parse time for the queries tested is significant compared to the query execution time. As shown in the table, the parse time is almost the same as the execution time for many queries and even larger for others. The last column in the table shows the percentage improvement in overall time (which includes parse time and execution time together). Pre-compiling allows for overall query execution time improvements that are quite significant, between 21% and 55% for the queries tested. Clearly, the execution time will scale with the size of the data set, but these results show how important removing parsing can be for performance improvement overall. For larger data sets of 1000 rows, the overall performance improvement is still about 10%. Given resource limitations on embedded devices, many queries on these devices execute on a small data set, and the performance improvement for these queries is dramatic. Further, the frequently executed INSERT statement has a near constant execution time that is similar in scale to the parse time and also benefits from pre-compiling.

5. CONCLUSIONS

In summary, pre-compiling query plans during the development process for embedded systems results in a significant improvement in code space utilization as well as reducing parsing times by up to 90% and execution times up to 50%. Developers still get all the advantages of working with a relational API and SQL on embedded systems without the overhead. This technique, although evaluated with LittleD, applies to any embedded database including Antelope and SQLite. Using pre-compilation allows relational database technology to be practically useful for all sizes of embedded devices including smart cards, Arduinos, and sensor nodes, and on-device data management improves efficiency and power utilization.

Future work includes adding typed placeholders into other clauses besides the WHERE clause, adding a task manager for network queries, and experimenting with SQLite to measure the benefits of pre-compiling queries for other systems.

6. ACKNOWLEDGMENTS

The authors acknowledge the financial support of NSERC.

7. REFERENCES

[1] N. Anciaux, L. Bouganim, and P. Pucheral. Memory Requirements for Query Execution in Highly Constrained Devices. VLDB '03, pages 694–705. VLDB Endowment, 2003.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. MDM '01, pages 3–14, London, UK, UK, 2001. Springer-Verlag.

[3] G. Douglas and R. Lawrence. LittleD: A SQL Database for Sensor Nodes and Embedded Applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 827–832, New York, NY, USA, 2014. ACM.

[4] D. K. Fisher and P. J. Gould. Open-Source Hardware Is a Low-Cost Alternative for Scientific Instrumentation and Research. *Modern Instrumentation*, 1(2):8–20, 2012.

[5] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[6] C. Severance. Massimo Banzi: Building Arduino. *Computer*, 47(1):11–12, Jan 2014.

[7] N. Tsiftes and A. Dunkels. A Database in Every Sensor. SenSys '11, pages 316–332, New York, NY, USA, 2011. ACM.