LittleD: A SQL Database for Sensor Nodes and Embedded Applications

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

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

ABSTRACT

Databases have reduced the cost associated with data management by abstracting applications from information processing challenges. There is an increasing need for managing and analyzing data in smaller embedded devices and sensor nodes. Due to resource limitations, these devices typically do not have well-defined data management APIs and standards such as the relational model and SQL. This results in increased complexity and cost. LittleD[1] is a SQL relational database allowing ad-hoc queries on sensor devices. The novel implementation of LittleD adapts to memory and code size restrictions by streamlining query parsing and execution and implementing efficient memory management techniques. Experimental results demonstrate that LittleD executes queries with joins and selections on devices with less than 2 KB memory in a few seconds.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query Processing

Keywords

Sensor node, microprocessor, embedded device, query, SQL

1. INTRODUCTION

Relational databases and SQL have reduced the cost and time to manage and analyze data. Increasingly, massive amounts of data are collected by small embedded and sensor devices and aggregated at servers for analysis. The ability to perform data analysis on device reduces latency and network transmissions which makes the devices more energy efficient and robust. Further, data collection may occur without network communication. As such, adapting database technology to these smallest devices is critical.

Despite the need, there are few data management systems or APIs for embedded and sensor devices. The resource constraints of commonly used sensor nodes and embedded systems pose a serious challenge. The most constrained devices might have as little as 16 KB of ROM for compiled code, 2 KB of RAM, and less than 1 MB of stable storage. These resource limitations restrict data management approaches and eliminate common embedded databases such as SQLite that run on more powerful hardware such as smart phones. Systems and algorithms must also adapt to the storage characteristics of flash memory. Flash memory has asymmetric performance with writes taking an order of magnitude longer than reads.

Previous systems such as PicoDBMS [2] and TinyDB [9] implement only the query execution engine on the device and rely on external input to parse, translate, and optimize the query plan to run on the device. The only system that has a near-SQL API is Antelope [10] which uses a similar but non-SQL compliant language to enable ad-hoc querying and execution. Antelope is designed to work with Contiki [4] and has some restrictive query processing limitations.

The contribution of this work is a SQL-compliant query engine that runs on devices with as little as 2 KB of memory. The distinctive features of LittleD are:

- A novel hand-written parser to minimize memory consumption.
- A query translation system that builds a query execution plan directly while parsing without the overhead of building parse and logical query trees.
- A guaranteed, fixed memory allocation system that performs query execution with a specified memory size while maintaining performance.
- The ability to perform the join of two or more tables, using filtering predicates and indexes where possible.
- Support for general expression evaluation.

LittleD has been experimentally evaluated by simulation using MSPSim on a Zolertia Z1 device and compared to Antelope [10]. Experimental results demonstrate that LittleD consistently uses less memory and is more flexible in query processing, with a higher degree of SQL compliance. LittleD achieves a resource efficiency that is vastly superior than that of Antelope.

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

[1]`https://github.com/graemedouglas/LittleD`.

2. BACKGROUND

Databases store, retrieve, and transform data. Relational database management systems (DBMSs) provide Atomicity, Consistency, Isolation, and Durability (ACID) properties as well as a Structured Query Language (SQL) for schema and data definition, retrieval, and transformation. Ideally, such facilities would also be available for a relational DBMS for microprocessors and sensor nodes.

Relational databases using SQL translate a user-specified query into an executable sequence of steps (see Figure 1). The parser tokenizes a query string to build a parse tree and then converts this parse tree into a logical query plan. An optimizer converts the logical query plan into an optimized query plan which is then translated into a physical (or executable) query plan (see Figure 2). Operators are inner nodes within the tree and scans of relations are leaves. The evaluator then executes the program to produce the results.

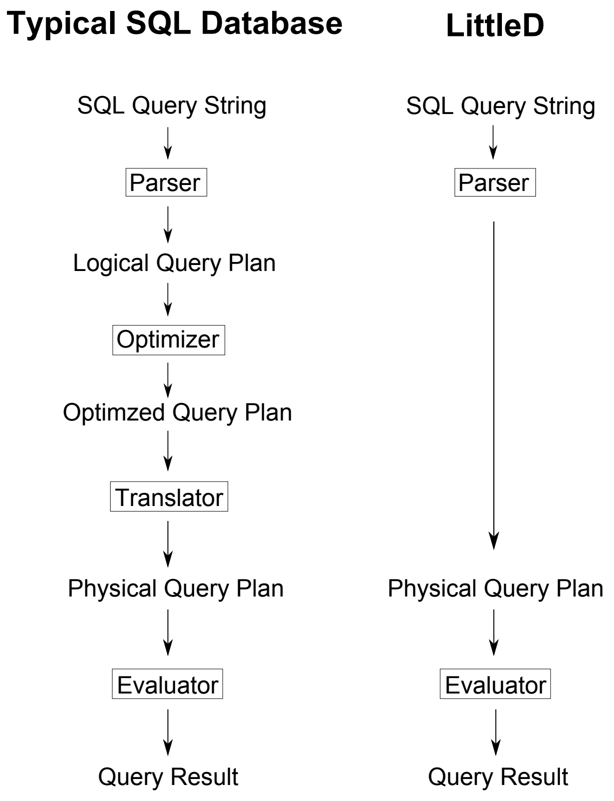


Figure 1: Query Processor Architecture for a typical SQL database and LittleD

```
SELECT R.x, T.y, T.a+T.b FROM R, T WHERE R.x=T.y
```

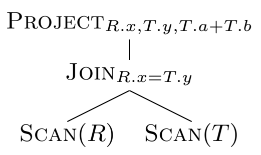


Figure 2: An example executable query plan.

For resource constrained environments, this query execution process is infeasible due to memory usage and code size constraints. Embedded devices may only have 16 to 128 KB of code space, and complicated components like the optimizer cannot be implemented. Further, devices may have as little as 2 KB of RAM. The generation of intermediate representations such as the logical and optimized query plans consume too much space. Consequently, systems are forced to make compromises on the components implemented and the level of SQL support.

One of the most common trade-offs is to perform only query execution on the device. In this case, a query is specified using a SQL-like language on a workstation, and then optimized and translated to an executable plan on the workstation. The executable plan is transmitted to the embedded device to execute. Although this eliminates a considerable amount of code complexity 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.

Systems such as COUGAR [3] and TinyDB [9] are distributed data systems intended to manage information over many networked sensors. These systems perform query parsing and translation off-device. A lead system exists which manages queries across the network.

Another technique is to simplify the database execution algorithms used and re-compute as much as possible. PicoDBMS [2] provides a framework for designing RAM-lower bound query execution algorithms. The authors demonstrate that confining algorithms to the absolute minimum RAM possible, even with available indexes, results in significant re-computation. However, increasing the RAM provided to these algorithms in even modest quantities drastically improves the algorithms' performance. PicoDBMS is designed for devices such as smart cards which have EEPROM and about 1 KB of memory.

For larger mobile devices, such as smart phones, embedded database products exist. SQLite[2] is a popular embedded database. Other similar SQL DBMSs include the H2 Database Engine[3], SQL Server Compact[4], and Mini SQL[5]. None of these systems target the most constrained of devices, but instead target more powerful platforms like mobile phones. Non-relational alternatives include BerkeleyDB[6] and UnQLite[7]. For example, SQLite requires at least 200 KB of code space and cannot run on popular platforms such as the Arduino.

For sensor devices, the most capable SQL-like database available is Antelope [10]. Antelope provides a query parser and execution system that can run on small embedded devices and a query language called AQL which is similar to SQL. AQL splits statements into smaller chunks to minimize memory usage. Data-retrieving queries are written in a more explicit form. For example, joins are given their own command. While this allows for shorter queries, it is also less flexible and does not conform to the SQL standard.

This work focuses on query processing on constrained devices. Related issues, including those of data persistence

[2]`http://www.sqlite.org/`
[3]`http://www.h2database.com/`
[4]`http://www.microsoft.com/en-us/sqlserver/editions/2012-editions/compact.aspx`
[5]`http://www.hughes.com.au/products/msql/`
[6]`http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html`
[7]`http://www.unqlite.org/`

and indexing on flash memories, have been widely studied including algorithms for updating data on flash memories [7], and efficient indexing structures optimized for wear-leveling and flash access patterns [1], [8]. The focus of this work is on the entire query engine and not specific implementations of indexing or flash-aware algorithms.

3. LittleD OVERVIEW

LittleD implements a SQL relational database execution engine capable of processing queries on resource-constrained devices. LittleD performs SQL parsing, translation, and optimization in a single component (Figure 1) that greatly reduces the memory-overhead for building query execution plans. As embedded devices have strict limits on memory, and static memory allocation is preferable over dynamic memory allocation, LittleD will execute a query given a statically-assigned memory buffer. All query planning and execution is performed using this static memory buffer, and memory allocation during query execution is tightly controlled and planned to maximize memory use efficiency. The programmer simply passes LittleD a byte-array to use.

LittleD implements SQL `SELECT-FROM-WHERE` syntax. A summary of the features implemented in comparison with Antelope is in Table 1. Each query string is stored as part of the compiled code or generated by the programmer dynamically. By eliminating traditional query translation, implementing relational operators as iterators, and using a stack-based memory allocator, LittleD attains efficient RAM usage and query performance.

The algorithms for each relational operator were restricted to single-pass or "tuple-at-a-time" approaches. SCAN, SELECTION, and PROJECTION all require only one tuple in memory at a time. There are two JOIN implementations: nested-loop join and index-nested loop join. Both algorithms require only one tuple of each relation be memory-resident at a time. SORT performs a single-record selection sort over some relation. This operator is used when an index does not exist for the sort attribute.

General expression evaluation is supported for common arithmetic, bitwise, and logical operators. The system uses a compact postfix notation to represent parsed expressions in the form of byte-code. A shunting-yard style algorithm is used to parse expressions [6]. The evaluation procedure consists of multiple steps. Each separate node in an expression is added to a temporary stack. When an operator is encountered, inputs are automatically converted to a common type, as is done in all relational databases. For example, if $1 + 2.3$ is the expression being performed, 1 must be converted to a floating point value. After type promotion, the partial expression rooted at the operator is evaluated and its result fed back onto the top of the temporary stack. This proceeds until the root operator or function has been evaluated.

LittleD is implemented using the C programming language. The system provides a compile-time configuration file so developers can choose which system features they need. The system compiles on systems supporting the GNU C Compiler or LLVM group of compilers including Linux, Windows, and Mac OS X, as well as for several devices the Contiki project supports. All relation schema information is stored in secondary storage with each relation.

3.1 Query Translation

Query parsing and optimization is memory and code space intensive. The use of a parse tree creates a number of challenges for highly constrained systems. An auto-generated lexer from existing tools is impossible to leverage since the generated code compiles to too large of a binary to be useful. The parse tree itself uses up memory. This extra RAM consumption puts the system at risk of running out of space for a query that it could otherwise execute. For this reason, the lexer and parser were manually written to convert SQL directly into executable byte-code.

Since query parsing, validation, and optimization are all combined into a single component, it must generate the best possible execution plan without complete information of the query. The idea is to generate the plan while scanning the input SQL string. This creates a unique set of challenges which are handled by applying common optimization heuristics to build quality, but not guaranteed optimal, plans. The heuristics and transformations applied are described below.

- Left-deep join trees are exclusively used as they generally provide better performance than right deep trees or bushy trees. Standard unindexed join algorithms iterate over tuples in the right relation while pulling each tuple from the left once. If the right child of a join is itself a join, it will need to be computed several times. As such, the right relation of each join operator is set to be a physical scan operator, so that re-computation can be avoided.

- Where possible, unindexed joins are upgraded to indexed joins. An indexed join uses one un-ordered relation to pull single tuples, and based on its filtering predicate, searches the other relation, assumed to be indexed.

Using these transformations provides the basis needed to build executable query plans. Given a `FROM` clause in a query, the corresponding scan operators for each of the physical relations can be built in order and then the joins can be initialized in reverse. For example, "`FROM` R_1, R_2, ... , R_n" is scanned in left to right order. Once memory space for each join has been allocated, the join between R_{n-1} and R_n is initialized followed by the join between R_{n-2} and R_{n-1}. Once all the joins have been initialized and then the expressions parsed and verified, joins that can use indexes are upgraded to do so. An example query translation is in Figure 3.

Example structures for expression nodes are in Figure 4. Sizes for these and other structures are provided in Table 2. Note that the SCAN, JOIN, and PROJECT operators may also generate some in-memory relation meta data. This meta data uses $2+4*$(number attributes)$+$(attribute name sizes) bytes. Tuple sizes are schema and platform dependent. A two integer record on the Z1 requires 5 bytes.

3.2 Memory Allocation

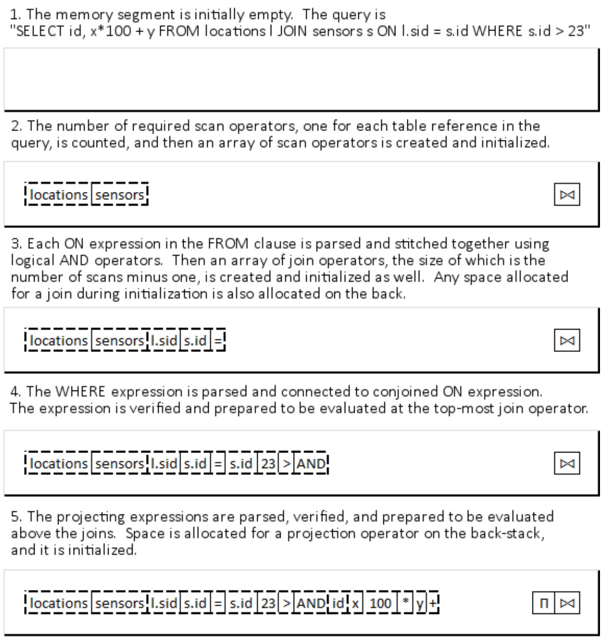
Memory allocation must be done as compactly as possible. To avoid fragmentation, LittleD uses a custom memory manager using two-stacks contained within one section of memory. The front stack's top starts at the beginning of the memory section, while the back stack's top is initially located at the end. Allocations always occur at the top of one of the two stack positions. While the LittleD memory manager never allows allocation to occur in any location

Table 1: Relational Operator Support and Implementation Details

Operator	LittleD	Antelope
JOIN	Nested tuple and index-based	Index-based
SELECT	Single-pass	Single-pass and index-based
PROJECT	Arbitrary expressions	Only simple attributes
SORT	Single-record selection sort on arbitrary expression	Index-based

Table 2: Fixed costs of common structures on Zolertia Z1.

Structure	Description	Size in memory
`db_exprnode_t`	Expression node supertype	2 bytes
`db_exprnode_attr_t`	Attribute node	6 bytes
`db_exprnode_dbint_t`	Integer value node	4 bytes
`db_exprnode_dbstring_t`	String value node	4 bytes + string size
`scan_t`	SCAN operator	24 bytes
`select_t`	SELECTION operator	8 bytes
`ntjoin_t`	Nested-loop JOIN operator	20 bytes
`project_t`	PROJECTION operator	10 bytes


1. The memory segment is initially empty. The query is "SELECT id, x*100 + y FROM locations l JOIN sensors s ON l.sid = s.id WHERE s.id > 23"

2. The number of required scan operators, one for each table reference in the query, is counted, and then an array of scan operators is created and initialized.

3. Each ON expression in the FROM clause is parsed and stitched together using logical AND operators. Then an array of join operators, the size of which is the number of scans minus one, is created and initialized as well. Any space allocated for a join during initialization is also allocated on the back.

4. The WHERE expression is parsed and connected to conjoined ON expression. The expression is verified and prepared to be evaluated at the top-most join operator.

5. The projecting expressions are parsed, verified, and prepared to be evaluated above the joins. Space is allocated for a projection operator on the back-stack, and it is initialized.

Figure 3: Example of LittleD's query translation process

```
/* Base type in expression bytecode. */
typedef struct
{
        db_uint8        type;
} db_exprnode_t;

/* Bytecode node for an attribute. */
typedef struct
{
        db_exprnode_t  base;
        db_uint8       attrpos;
        db_uint8       whichtable;
} db_exprnode_attr_t;

/* Bytecode node for an integer value. */
typedef struct
{
        db_exprnode_t  base;
        db_int         integer;
} db_exprnode_int_t;
```

Figure 4: Example expression nodes for LittleD

other than the top of a stack, memory segments may be freed from places that are not on the top. Due to the flow of data through an executable query plan, memory allocations and de-allocations tend to occur in a fixed order. The overhead costs to such a design are minimal, costing the equivalent of two pointers per allocated segment on the front and one pointer per allocated segment on the back.

Another advantage to using this memory management strategy is that it is trivial to detect and handle out-of-memory errors on the fly. It is also possible for the system to detect out of memory errors long before they occur by directly calculating the memory required to execute a physical query plan after it is built. This could potentially save valuable execution cycles for more complicated queries the system does not have the resources to answer. More importantly, it prevents segmentation faults from happening, which could lead to data loss, device restarting or other destructive, unpredictable behaviour.

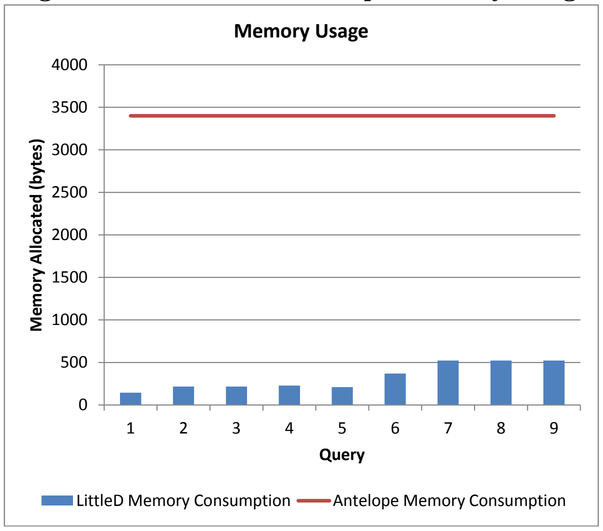
3.3 Indexing

The indexing method used by LittleD is the same as Antelope's inline indexing [10]. It uses $O(1)$ additional space and assumes that the relation is physically stored in monotonically increasing order of the key. This is a common occurrence for time series data collected using sensors. Equality joins over such an attribute allow for binary search of the indexed relation to find matching tuples and then simple in-order reads until a non-matching tuple is encountered. Implementing other indexing methods is future work.

4. EXPERIMENTAL RESULTS

LittleD and Antelope were compared using the MSPSim simulator[8] on a simulated Zolertia Z1 device. The queries evaluated for LittleD and Antelope, as expressed in SQL and AQL respectively, are given in Table 3. Execution time and memory results were collected for LittleD. Antelope has a fixed cost of 3.4 KB of RAM of static allocations along with a variable amount of dynamic allocations, between 0 and 328 bytes [10]. In comparison, LittleD is given a fixed amount of memory to allocate from. Less than 1 KB of memory was required for each query executed. This means that for the most memory intensive queries executed, Antelope uses at least 500% more memory than LittleD. For the least memory intensive query, this disparity grows to over 1300%. This is extremely important as memory is the most limited component on embedded devices. LittleD has extremely low memory requirements that makes it possible to run queries on even the smallest devices.

Figure 5: LittleD vs. Antelope Memory Usage



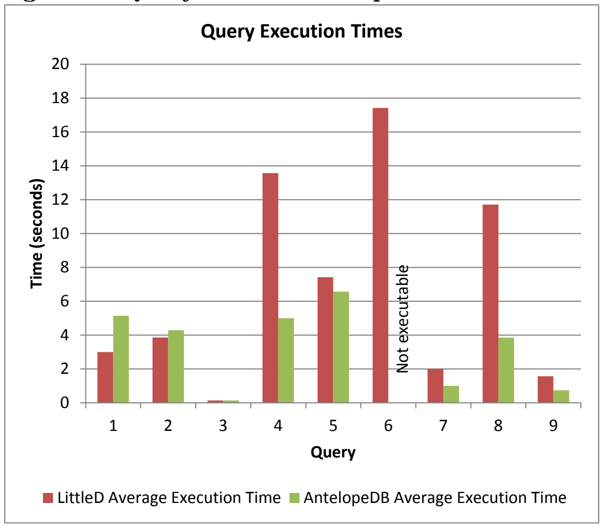
Both systems use inline indexes on the first attribute for all tables and give preference to using the right relation for indexing a join whenever possible. Although no benchmarks are provided here, it is also possible to execute unindexed joins using LittleD, while it is not possible for Antelope (see Table 1).

Both systems require considerable code space. When compiled for the Z1 device, Antelope required about 48 KB, while LittleD required approximately 55 KB. Both code space sizes include a small amount of utility code to generate data and run benchmarks. The amount of extra code used is comparable to that needed to collect data in a real world application.

Query 1 is a raw, serialized scan over a large relation, and LittleD outperforms Antelope in this task because LittleD does not require projections when selecting all data, whereas Antelope does.

Both databases exhibit nearly identical times for queries 2 and 3. This is attributed to the use of the indexes to significantly reduce the number of tuples processed. For query 4, Antelope uses an index, while LittleD currently has more limited index support. When no index is used, the two databases perform nearly identically for SELECTIONS, as query 5 shows.

Figure 6: Query Execution Experimental Results



Query 6 highlights that the time required to execute a query is approximately linearly proportional to the number of calculations in all of the expressions. Despite the small output relation size, the cost of calculating the result of three operations over each tuple drastically affects the execution speed. Further, this query highlights LittleD's ability to project arbitrary expressions, whereas Antelope is only able to project individual attributes.

Queries 7, 8, and 9 show that Antelope outperforms LittleD on similar joins. However, Antelope tacitly assumes that all joins are equijoins involving single attributes of the same name, while LittleD can evaluate an arbitrary join expression. This allows LittleD to have greater flexibility in how indexes can be leveraged, at the cost of execution speed; Antelope's joins are approximately two to three times as fast as LittleD's. Furthermore, our experimentation revealed errors in Antelope's joins that affected its correctness. For the test cases presented, the number of results generated was correct, but for a variety of other joins the output was incorrect. Given the limitations in the Antelope join implementation, it is difficult to have a meaningful comparison. Query 7 is faster than query 8 for both systems because it performs fewer binary searches on the indexed relation.

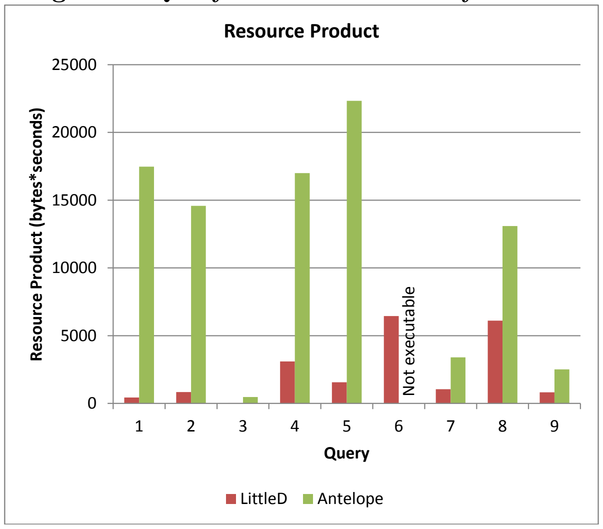
A unified measure of performance is derived by multiplying execution time by memory consumed. We call this the *resource product*, and it indicates how well the database balances memory usage and execution time. The lower the resource product, the better the database uses its resources. LittleD has a smaller resource product for all queries tested. The best case for Antelope is query 8 where its resource product is twice as large as LittleD. In the worst case, Antelope's product is over 40 times larger. LittleD is the most memory efficient, which is the most constrained resource, while maintaining comparable query time performance and providing more general SQL query capabilities.

[8]This is the cycle-accurate simulator used by Contiki for testing. `https://github.com/mspsim/mspsim`

Table 3: Queries Executed

Query	LittleD SQL String	Antelope AQL String	Relation Info	Result Size
1	`SELECT * FROM r`	`SELECT * FROM r`	$\lvert r\rvert = 10000$	10000
2	`SELECT * FROM r WHERE attr0 > 4999`	`SELECT attr0, attr1 FROM r WHERE attr0 > 4999`	$\lvert r\rvert = 10000$	5000
3	`SELECT * FROM r WHERE attr0 < 100`	`SELECT attr0, attr1 FROM r WHERE attr0 < 100`	$\lvert r\rvert = 10000$	100
4	`SELECT * FROM r WHERE attr0 >= 7 AND attr0 <= 6009`	`SELECT attr0, attr1 FROM r WHERE attr0 >= 7 AND attr0 <= 6009`	$\lvert r\rvert = 10000$	6003
5	`SELECT * FROM r WHERE attr1 < 100`	`SELECT attr0, attr1 FROM r WHERE attr1 < 100`	$\lvert r\rvert = 10000$	6003
6	`SELECT attr0, attr2/4, (attr1+1)*2 FROM r WHERE attr1 < 10 OR (attr2 / 4) = 13`	*No Executable Equivalent*	$\lvert r\rvert = 10000$	6
7	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`	$\lvert l\rvert = 100, \lvert r\rvert = 1000$	100
8	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`	$\lvert l\rvert = 1000, \lvert r\rvert = 100$	100
9	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`	$\lvert l\rvert = 100, \lvert r\rvert = 100$	100

Figure 7: Query Resource Efficiency Results



5. CONCLUSIONS

This work presents LittleD, a relational SQL database for the smallest embedded devices. LittleD is feature and performance competitive with Antelope while being more flexible and using drastically less memory for query processing. Future research will focus on indexing and algorithmic improvements to relational operators to lower RAM and energy usage. This will include the implementation of MinSort [5] to greatly improve unindexed sorting speeds. Different parsing techniques, from off-device query compilation to new representations for SQL-like languages, will be explored to reduce the code space requirements of a complete system. Further work will be done to use this system on common platforms such as the Arduino[9].

[9]`http://www.arduino.cc/`

6. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, Aug. 2009.

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

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

[4] Contiki Operating System. `http://www.contiki-os.org/`. Accessed: 2013-08-15.

[5] T. Cossentine and R. Lawrence. Fast Sorting on Flash Memory Sensor Nodes. IDEAS '10, pages 105–113, New York, NY, USA, 2010. ACM.

[6] E. W. Dijkstra. Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical Report 35, Mathematisch Centrum, Amsterdam, 1961.

[7] E. Gal and S. Toledo. Algorithms and Data Sructures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[8] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. μ-tree: An Ordered Index Structure for NAND Flash Memory. EMSOFT '07, pages 144–153, New York, NY, USA, 2007. ACM.

[9] 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.

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