LittleD: A Relational Database Management System for Resource Constrained Computing Devices

by

Graeme Robert Douglas

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

B.SC. COMPUTER SCIENCE HONOURS

in

THE IRVING K. BARBER SCHOOL OF ARTS AND SCIENCES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2014

© Graeme Robert Douglas, 2014

Abstract

Database systems have greatly reduced the cost of managing large collections of data. The use of small computing devices for collecting data, such as embedded microprocessors and sensor nodes, has greatly increased. Due to their limited resources, adequate database systems do not currently exist for the smallest of computers. LittleD is a relational database supporting ad-hoc queries on microprocessors. By using reduced-footprint parsing techniques and compact memory allocation strategies, LittleD can execute common SQL queries involving joins and selections within seconds while requiring less than 2KB of memory.

Table of Contents

Abstract . . . . **ii**

Glossary of Notation . . . . **vi**

Chapter 1: Introduction **1**
- 1.1 Motivation 1
- 1.2 Challenges 1
- 1.3 Goals and Research Questions 2
- 1.4 Thesis Statement 2

Chapter 2: Background **3**
- 2.1 The Relational Model and its Algebra 3
- 2.2 SQL and its Core Features 6
- 2.3 Existing Query Processing Techniques 13
- 2.4 Previous Work 17

Chapter 3: LittleD Implementation Summary **19**

Chapter 4: Experimental Results and Discussion **23**
- 4.1 Experimental Setup 23
- 4.2 Code Space and Memory Requirements 23
- 4.3 Discussion of Results 24

Chapter 5: Conclusion **29**
- 5.1 Future Work 30

Bibliography **31**

List of Tables

Table 3.1 Relational Operator Support and Implementation Details 19

Table 4.1 Queries Executed 27
Table 4.2 Information about queried relations. For joins, table l has same schema as table r. 28

List of Figures

Figure 2.1 A relation in Codd's model, called "employee". 3
Figure 2.2 A relation in Codd's model, called "department". . . 3
Figure 2.3 A relation in Codd's model, called "costs". 4
Figure 2.4 Example of a selection 4
Figure 2.5 Example of a projection 5
Figure 2.6 Example of a Cartesian product 5
Figure 2.7 Example of a θ-join 6
Figure 2.8 Syntax for creating a relation in SQL. 7
Figure 2.9 An example relation with key and nullity constraints. 8
Figure 2.10 Syntax for adding a tuple to a relation. 8
Figure 2.11 Syntax for deleting tuples from a relation. 9
Figure 2.12 Syntax for updating tuples in a relation. 9
Figure 2.13 Syntax for extracting data from a database in SQL. . 10
Figure 2.14 A `SELECT` statement without a projection. 10
Figure 2.15 A `SELECT` statement with projecting expressions. . . 10
Figure 2.16 A `SELECT` statement involving two implicit joins. The condition for the joins are all specified in the `WHERE` clause. 11
Figure 2.17 A `SELECT` statement involving one implicit join and two explicit joins. Note that the join condition can be specified in either the `ON` clause or the `WHERE` clause for explicit join notation. 11

Figure 3.1 Query Processor Architecture for a typical SQL database and LittleD 20

Figure 4.1 Query Execution Experimental Results 24
Figure 4.2 LittleD vs. Antelope Memory Usage 25
Figure 4.3 Query Resource Efficiency Results 26

Glossary of Notation

σ Relational Selection Operator. 4
π Relational Projection Operator. 4
$\times$ Relational Cartesian Product Operator. 4
$\bowtie$ Relational Join Operator. 4

Chapter 1

Introduction

1.1 Motivation

A growing push towards the collection of data from any and all processes is underway. The movement known as the "internet-of-things" [GBMP13] is networking together devices measuring and controlling home appliances, industrial mechanisms, transportation systems, environmental sensors, and many other previously disconnected entities. Gathering and leveraging this massive amount of information requires cheap and abundant sensor nodes or micro-controllers. In a typical use scenario, sensor nodes may be used to collect data over an extended period of time. Periodically, the data will be retrieved from each sensor at which point analysis is performed on the complete data collection. This analysis may involve filtering irrelevant information away, transforming data according to some mathematical procedure, and computing statistics such as means or extremums over data sets.

Should each device contain significant amounts of data, the time and energy costs of transmitting the information across networks can be prohibitive. Further, network communications endanger collected data by exposing them to imperfect or potentially unreliable protocols. On-device querying allows for reduced network communication, thus reducing potential data loss or corruption and energy consumption. Further, data collection may occur without the presence of network communication, and it may be necessary for a device to make autonomous decisions based on previously collected information. Thus developing data query and management techniques for the smallest of computing devices is critical.

1.2 Challenges

The cost of deploying even simple data management solutions for small computing devices remains high. This is not without good reason. A typical database management system (DBMS), targeting a workstation or server, may have gigabytes or even petabytes worth of random access memory for

query translation, optimization, and execution. A typical sensor node has 1 KB to 1MB of RAM, with many having between 2KB to 64KB of memory. Further, these devices may be expected to collect data on a battery over days, weeks, or months, so energy must be used sparingly. Stable storage usually comes in the form of flash memory, either of NAND or NOR type [GT05]. Finally, code space, which is the memory reserved for compiled execution instructions, is usually in the range of 16 KB to 1 MB.

Existing APIs provide access to the most popular relational database software for servers. Significant research and development effort has been spent optimizing query performance for these applications. A server could potentially have millions of times more memory than an embedded processor, in addition to effectively unlimited energy and copious stable storage space for both compiled code and data management. These systems are intended to manage information at a scale not possible with the typical microprocessor.

1.3 Goals and Research Questions

The primary goal of LittleD is to provide an application-ready relational database management system for the smallest of computing devices. The developed database should seek to best utilize the limited resources available, including memory, processing cycles and energy, code space, and stable storage, as efficiently as possible.

An important subgoal of this thesis work is to determine whether SQL translation is possible on device. Query execution algorithms have been provided in previous works [ABP03] for sensor nodes, but can SQL translation techniques be leveraged concurrently to form a complete system? What restrictions must be respected? It is expected that execution cycles and memory will oppose each other, but how will code space play into the resource mix? What relational operators can and should be implemented on device, and which relational operators are potentially less beneficial?

1.4 Thesis Statement

This thesis will examine if it is possible to build a SQL query processor on an embedded sensor node that has efficient performance for querying and data management. The expectation is that a relational database system will be constructed, and the major limitations of memory, CPU, and code space will be analyzed to determine the most critical resource constraints. The system will be compared to Antelope [TD11].

Chapter 2

Background

2.1 The Relational Model and its Algebra

Codd [Cod70] first proposed the relational model in 1970, and though today's relational databases differ from the original model slightly in some details, relational database management systems (DBMSs) continue to be popular. In the relational model, a number of operators are defined which operate over relations, which are defined mathematically. Let $S_1, S_2, S_3, \ldots, S_n$ be a collection of n sets. A *relation* R is a subset of the Cartesian product $S_1 \times S_2 \times S_3 \times \cdots \times S_n$. That is, R consists of n-tuples, sometimes referred to as *records* or *rows*, where the ith element in a tuple is an element of S_i. A relation that contains n-tuples is said to be n-ary. As is customary for sets, the order of tuples in a relation is irrelevant, and each tuple is unique in that there are no duplicate tuples. As implied by the Cartesian product, the order of elements in the tuple is important. Typically, each element in a tuple, called an *attribute* or *column*, is labeled with a name to convey some meaning of the domain it is selected from.

Figure 2.1: A relation in Codd's model, called "employee".

employee	**Name**	**depNum**	**Salary**
	`'Sarah Oliver'`	1	$100000
	`'John Smith'`	1	$45000
	`'Christian Elliott'`	2	$30000
	`'Taylor Moore'`	3	$80000
	`'Jen Austin'`	3	$90000

Figure 2.2: A relation in Codd's model, called "department".

department	**ID**	**Name**
	1	`'Management'`
	2	`'Development'`
	3	`'Sales'`

Figure 2.3: A relation in Codd's model, called "costs".

costs	**invoiceNum**	**depNum**	**Description**	**Amount**
	1	1	'Training for John'	$3000
	2	1	'Logo design work'	$9000
	3	3	'Sales tracking app'	$10000
	4	3	'Sales retreat'	$1000
	5	2	'IDE software'	$8329

Note that this construction for relations bears a strong similarity to a table of values. The attributes are the columns within the table, and the tuples are the rows.

The power of the relational model comes from the operators defined, which taken together form the *relational algebra*. Operators can be classified as either *unary*, those which take one relation as its input, or *binary*, those which take two relations as input. Every operator returns a single relation as output. The following is a brief discussion of relational operators.

The selection operator σ is a unary operator that also takes as input a Boolean predicate p involving attributes of the relation. The result of a selection is a new relation that contains only those tuples of the input relation that satisfy p. One can imagine that this operator iterates through each tuple in a relation, and allows a tuple to pass through into the output relation if it satisfies the predicate. The tuples present in the output relation are not changed otherwise, though the number of tuples in the output relation may decrease.

Figure 2.4: A selection: $\sigma_{\texttt{Salary}>=50000 \text{ AND } \texttt{Salary}<100000}(employee)$

Name	**depNum**	**Salary**
'Taylor Moore'	3	$80000
'Jen Austin'	3	$90000

The projection operator π is a unary operator that additionally takes as input a series of expressions that describe how to transform each tuple in the input relation into a new tuple to be placed in the output relation. A simple example would be creating a new relation from an input relation R by adding attributes x and y of R together, as well as including c as the second attribute in the output. This would be expressed as $\pi_{a+b,c}(R)$. One should notice that the number of attributes in the tuples of the output relation may be less than, equal to, or greater than the number of attributes in the tuples

from the input relation. Further, the order of the attributes could change.

Figure 2.5: A projection: $\pi_{\texttt{invoiceNum, Amount}}(employee)$

invoiceNum	Amount
1	$3000
2	$9000
3	$10000
4	$1000
5	$8329

The Cartesian product, $\times$, is a binary operator which takes as input two relations R and S and outputs a relation O. The tuples of O are constructed by taking all pairs of tuples between R and S and creating new tuples where all attribute values from each pair exist in the new tuple. Written $R \times S$, a tuple in O has the form $(a_1, a_2, \ldots, a_n, a_{n+1}, a_{n+2}, \ldots, a_{m+n})$ where a_i appears in a tuple from R if $0 \leq i \leq n$ and appears in S if $0 \leq i \leq m$.

Figure 2.6: A Cartesian product: *employee* $\times$ *department*

Name	depNum	Salary	ID	Name
'Sarah Oliver'	1	$100000	1	'Management'
'Sarah Oliver'	1	$100000	2	'Development'
'Sarah Oliver'	1	$100000	3	'Sales'
'John Smith'	1	$45000	1	'Management'
'John Smith'	1	$45000	2	'Development'
'John Smith'	1	$45000	3	'Sales'
'Christian Elliott'	2	$30000	1	'Management'
'Christian Elliott'	2	$30000	2	'Development'
'Christian Elliott'	2	$30000	3	'Sales'
'Taylor Moore'	3	$80000	1	'Management'
'Taylor Moore'	3	$80000	2	'Development'
'Taylor Moore'	3	$80000	3	'Sales'
'Jen Austin'	3	$90000	1	'Management'
'Jen Austin'	3	$90000	2	'Development'
'Jen Austin'	3	$90000	3	'Sales'

Join operators are binary operators that produce an improper subset of the Cartesian product over the same input relations. Within the relational algebra, many types of joins are defined, but for the present work, only one is necessary. A θ-join, θ-$\bowtie$, filters the result of a Cartesian product by a

predicate θ. The following is an identity in terms of the resultant relation:

$$\bowtie_\theta (R, S) = \sigma_\theta(R \times S) \tag{2.1}$$

Figure 2.7: A θ-join: $employee \bowtie_{employee.deptNum=department.ID} department$

Name	depNum	Salary	ID	Name
'Sarah Oliver'	1	$100000	1	'Management'
'John Smith'	1	$45000	1	'Management'
'Christian Elliott'	2	$30000	2	'Development'
'Taylor Moore'	3	$80000	3	'Sales'
'Jen Austin'	3	$90000	3	'Sales'

In contrast to the theoretical model of relations and their algebra that has been developed, practical considerations dictate deviations in how the model is implemented. It is sometimes necessary to allow for a relation to contain duplicate records. Often times, database systems will allow for ways to remove duplicates within a relation. While most relational DBMSs will not guarantee a specific order on un-indexed relations, it is reasonable to assume that an implicit order exists due to the nature of digital storage media. Further, a computer is generally unable to determine the domain of possible values from the name of an attribute. As such, schemas, which are ordered lists of name and domain type pairs for each attribute, are used. A schema may also describe other properties of a relation not discussed here, including value constraints.

2.2 SQL and its Core Features

How a user of a relational database interacts with the system is through SQL, which stands for Structured Query Language. SQL has been standardized as a language[1] by the American National Standards Institute[2] first in 1986 and then by the International Organization of Standardization[3] in 1987. Enterprise class relational DBMSs implement many of the numerous standardized language features found in SQL. For databases targeting smaller computing devices, such robust language support is impossible due to code

[1] http://standards.iso.org/ittf/PubliclyAvailableStandards/c053681_ISO_IEC_9075-1_2011.zip

[2] http://www.ansi.org/

[3] http://www.iso.org/iso/home.html

space restrictions. What immediately follows is a discussion of the core set of SQL features supported by this work.

In SQL, *identifiers* are used to name relations and attributes, as well as other important objects such as indexes. *Reserved words* or *key words* are strings that cannot be used as identifiers unless identifier quotes are used as delimiters. Commands such as `SELECT` and `WHERE` are examples of reserved words. SQL is written in *statements*, which are sets of tokens, including reserved words and identifiers, that should be interpreted and executed as a whole. Statements are whitespace ignorant except for the whitespace needed to delimit individual tokens. In general, statements are classified by the action they perform. All statements have a form or *syntax* they must adhere to in order to be valid.

Schemas for relations are defined in CREATE TABLE statements. They follow the basic form found in Figure 2.8. The attributes are defined in the

Figure 2.8: Syntax for creating a relation in SQL.

```
CREATE TABLE <relation-name>
(
        <attribute1-identifier>
                <attribute1-type>
                [<attribute1-constraints>],
        <attribute2-identifier>
                <attribute2-type>
                [<attribute2-constraints>],
        ...
        <attributeN-identifier>
                <attributeN-type>
                [<attributeN-constraints>]
        [, extra_constraints]
);
```

order they should appear in the tuples of the relation. The most common types of attribute constraints are *key constraints*. A *superkey* for a relation is any subset of the attributes that uniquely identify a row. A *key* is a superkey where no attribute can be removed from the set such that the set is still a superkey. A *primary key* is a key that is designated to identify tuples within a relation. A *foreign key* is a key within a relation R that identifies tuples in a relation S. The attributes of a foreign key from R to S each map directly to a single attribute of the primary key of relation S. Further, the

domains of the attributes in the foreign key must be an improper subset of the domain of the attribute each is mapped to.

Another commonly used constraint involves the nullity of an attribute. `NULL` represents a lack of value or a missing value. An element in a tuple may be `NULL` regardless of the attribute's domain provided no constraint prevents it from doing so. If `NOT NULL` is included in the list of constraints for an attribute in a relation's schema, that attribute can never be set to `NULL`.

Figure 2.9: An example relation with key and nullity constraints.

```
CREATE TABLE department
(
        id INT PRIMARY KEY,
        name CHAR NOT NULL
);
```

Once a schema has been defined, `INSERT`, `UPDATE`, and `DELETE` statements can be used to add data to a relation, modify data in a relation, and remove data from a relation, respectively. These three statements form what is referred to as the *Data Manipulation Language* portion of SQL. The syntax for an `INSERT` statement is given by Figure 2.10. In the second form

Figure 2.10: Syntax for adding a tuple to a relation.

```
INSERT INTO <relation-name>
VALUES (
        <attribute1-val>,
        <attribute2-val>,
        ...,
        <attributeN-val>
)
```

or alternatively

```
INSERT INTO <relation-name>
(<attribute-list>) VALUES (
        <attribute-value-list>
)
```

the values must agree with the domains specified in the list of attributes.

Any unspecified columns in the attribute list are set to `NULL`. If a nullity constraint is violated, an error would occur.

A `DELETE` statement removes tuples from a relation. Its syntax is as in Figure 2.11. The `WHERE` clause specifies a filtering predicate that selectively

Figure 2.11: Syntax for deleting tuples from a relation.

```
DELETE FROM <relation-name>
WHERE <predicate>
```

chooses tuples for removal. For instance, `DELETE FROM R WHERE x = 1` only removes those tuples in the relation R that have the x attribute set to 1.

The `UPDATE` statement modifies tuples already in a relation. The statement takes the form of Figure 2.12. The attribute values must match the

Figure 2.12: Syntax for updating tuples in a relation.

```
UPDATE <relation-name>
SET <attribute-name1>=<attribute-val1>
[, <attribute-name2>=<attribute-val2>[, ...]]
WHERE <predicate>
```

domains of the paired attribute names specified in the `SET` clause. As with `DELETE` statements, the `WHERE` clause selectively chooses which tuples to update. As an example, `UPDATE R SET x=2 WHERE y > 1` changes all x attributes in relation R to have value 2 whenever the attribute y has value greater than 1. In this example, x has some sort of real-valued domain.

The records stored within a database are useless if there is no way to extract them. The `SELECT` statement is used to retrieve information from one or more relations. The result of a `SELECT` statement is usually a relation, but can also be an atomic value. Codd [Cod70] defined an *atomic value* as one that could not be decomposed. An attribute's value is almost always atomic. The `SELECT` statement syntax is in Figure 2.13. The `SELECT` statement is both powerful and complicated. The only mandatory clause within the statement is the `SELECT` clause, but only the most trivial of queries requires it alone. The list of expressions in the `SELECT` clause may involve attributes from relations listed in the `FROM` clause. Each expression can be arbitrarily complex, involving arithmetic operations ($+$, $-$, $*$, $/$, $\%$),

Figure 2.13: Syntax for extracting data from a database in SQL.

```
SELECT <expression-list>
FROM <relation-list>
WHERE <predicate>
GROUP BY <grouping-list>
HAVING <predicate-list>
ORDER BY <expression-list>
```

defined functions (`SUBSTR(...)`), as well as aggregate functions. Further, the special * operator is shorthand for all attributes.

Figure 2.14: A `SELECT` statement without a projection.

```
SELECT *
FROM employee
```

Figure 2.15: A `SELECT` statement with projecting expressions.

```
SELECT name, salary*2 - 10000
FROM employee
```

The `FROM` clause contains a list of relations to be joined together. In the simplest of forms, tables are listed with commas separating each table, with each comma implicitly defining a Cartesian product or a join if an appropriate filtering predicate is specified in the `WHERE` clause. Figure 2.16 provides an example of such a query.

Alternatively, one can use an explicit syntax to specify joins. As a single relation specification within the `FROM` clause, a series of tables can be joined using `JOIN ON` syntax, as in Figure 2.17.

This same syntax can also be used to specify different types of outer joins not covered here.

The `WHERE` clause is familiar from `UPDATE` and `DELETE` statements. For `SELECT` statements, the `WHERE` clause is very important because query translators are able to break apart a predicate specified here such that each piece can be evaluated within the specified joins. This is important because it means tuples are filtered before they are considered for joins, which results in fewer computations for a query. Better yet, the user writing the query

Figure 2.16: A SELECT statement involving two implicit joins. The condition for the joins are all specified in the WHERE clause.

```
SELECT *
FROM R, S, T
WHERE R.id = S.id AND S.t_id = T.id
```

Figure 2.17: A SELECT statement involving one implicit join and two explicit joins. Note that the join condition can be specified in either the ON clause or the WHERE clause for explicit join notation.

```
SELECT *
FROM R,
  S JOIN T ON S.t_id = T.id JOIN W ON T.id = W.id
WHERE R.id = S.id
```

usually is not required to think about how to optimally filter tuples at each join. From the user's perspective, the FROM clause defines a relation resulting from repeated Cartesian products that are filtered using a selection predicate defined in the WHERE clause. As will be seen, there are code complexity costs associated with such optimization.

The GROUP BY clause controls how aggregate functions operate. An aggregate function takes as input some arbitrary inner expression not involving an aggregate function itself. According to the GROUP BY clause, the input relation (the result of the FROM clause's joins filtered by ON and WHERE predicates) is iterated over and the inner expression is evaluated once for every tuple, and an aggregate value is computed. Almost all SQL systems support AVG, MAX, MIN, COUNT, and SUM as aggregate functions, each doing exactly what the name implies.

Consider a relation $R = \{(1, 2), (3, 4)\}$ with the first attribute named *odds* and the second named *evens*. Then

```
SELECT MAX(odds) * MIN(evens) FROM R
```

would return the relation $\{(6)\}$ (or in many systems, the single atomic value 6).

In many instances, aggregates may need to be performed within partitions or categories. For example, a business owner may wish to know what his most expensive bill was in each department. The relation *costs* contains

attributes *depNum*, *invoiceNum*, *description*, and *amount* (see Figure 2.3). Simply specifying the query

```
SELECT MAX(amount) FROM costs
```

will result in getting the max overall cost. Using the `GROUP BY` clause, this becomes

```
SELECT depNum, MAX(amount)
FROM costs
GROUP BY depNum
```

which will return a max for each department. If the owner wanted to further break down costs by invoice, she may simply add the *invoiceNum* attribute to the `GROUP BY` list

```
SELECT depNum, invoiceNum MAX(amount)
FROM costs
GROUP BY depNum, invoiceNum
```

The attributes or expressions appearing in the `SELECT` clause that are outside of an aggregate function must be in the `GROUP BY` clause. Otherwise, the query will not make sense, and most systems will raise an error.

Filtering predicates involving aggregate functions can be specified in the `HAVING` clause, but not the `WHERE` clause. A predicate defined in the `WHERE` clause is always evaluated before any aggregate function in the `SELECT` clause is defined. This is for performance and correctness reasons. If an aggregate is being computed, the fewer tuples to be considered, the faster the computation completes. Further, an aggregate should not consider those tuples which do not match the `WHERE` predicate since it might affect the computed values. The `HAVING` clause evaluates after an aggregate is computed. Continuing the last example, if the business owner only wishes to view those costs above $8000 dollars, she would write

```
SELECT depNum, invoiceNum, MAX(amount)
FROM costs
GROUP BY depNum, invoiceNum
HAVING MAX(amount) > 8000
```

Once a relation has been built, it may be necessary to order the result according arbitrary criteria. The `ORDER BY` clause provides the necessary

mechanism to do so. The clause specifies a list of attributes or expressions, each optionally followed by a keyword to indicate ordering direction, either `ASCENDING` (`ASC`) or `DESCENDING` (`DESC`). `ASCENDING` is assumed if no direction keyword is present. Should the business owner wish to know which departments had the most expensive bills over $10000 in decreasing order, she could write

```
SELECT depNum, invoiceNum, MAX(amount)
FROM costs
GROUP BY depNum, invoiceNum
HAVING MAX(amount) > 8000
ORDER BY MAX(amount) DESC
```

Since order must be enforced after joins and aggregation, aggregate functions may appear in the `ORDER BY` clause but are not required. Further, aggregate functions used in the `ORDER BY` clause need not show up anywhere else. For instance, the business owner may wish to order the results primarily by the average money spent per department, and then secondarily by the department number. She would then write

```
SELECT depNum, invoiceNum, MAX(amount)
FROM costs
GROUP BY depNum
HAVING MAX(amount) > 8000
ORDER BY AVG(amount) DESC, depNum
```

Note that since $depNum$ appears in the `ORDER BY` clause, it must also appear in the `GROUP BY` clause. Otherwise, the query would be asking the database to sort on values that may not be consistent within the same group, and would have no way of deciding which value to choose. If there are no grouping attributes and no use of aggregate functions, then any attribute or expression may appear in the `ORDER BY` clause.

2.3 Existing Query Processing Techniques

Converting the text of a query to an executable form is called *query processing*, to which there are usually multiple steps [Gra93]. Query parsing converts the input query into a tree of symbols called an *expression tree* or *parse tree*. This parse tree is then translated into a *logical query tree* or *logical query plan* to be optimized. Once all optimization work is complete,

the resulting logical query tree is converted to a *physical plan*, also called a *physical query tree*. The evaluator then executes the physical plan to return the query output.

Query optimization is perhaps the most important step for most systems. A logical query plan can be improved to use less computational cycles, less memory, or both by using appropriate algorithms for operators as well as by reorganizing entire sections of the query tree. Perhaps the most common instance of such reorganization is expression pushing. One form of expression pushing involves breaking apart predicates and pushing pieces down into operators at lower portions of the tree such that fewer tuples are processed at operators higher in the tree. This form was discussed in Section 2.2. Another type of expression pushing places new operators, usually projections, at targeted areas in the query plan to reduce the size of each tuple processed.

Algorithm choice for the relational operators is another tool used by database systems to reduce query execution costs. Typical relational database systems have the luxury of large amounts of available RAM in addition to virtually unlimited stable storage. These system qualities allow for business-class RDBMSs to leverage algorithms involving hash buckets [DKO+84], external merge sorts [DKO+84], among other clever tools.

None of these techniques are feasible for databases targeting much smaller devices. Using valuable code space on complicated optimization procedures or varying implementations for each operator is impractical. Algorithms making heavy use of memory are not feasible since the database may have less than 1KB of RAM available to it. Swapping in-memory data to and from stable storage is also not advisable since flash memories have asymmetric read and write performance characteristics. Thus, new querying techniques are required.

The simplest technique, as discussed by the PicoDBMS [ABP03] framework, is to enforce minimal RAM usage through re-computation. The authors define a framework including algorithms which achieve the lowest RAM footprint possible for basic relational operators. The authors demonstrate that this re-computation becomes exorbitantly costly as the amount of data grows. Giving even small amounts of extra RAM to their algorithms greatly reduces execution cycles.

A similar class of algorithms for relational operators known as 'tuple-at-a-time' algorithms require only a single tuple or two be present in memory at a given instant. Each operator returns a single tuple on request. When a tuple is requested, it requests the next tuple from each of its child operators and attempts to produce a result. If an operator cannot produce a tuple from the current child tuples, then it will request new child tuples as necessary. When

any of the children run out of tuples to return, then the current operator has also run out of tuples. Scan operators, which do not themselves have children and are always the deepest child operators, read tuples into memory, one at a time, until the entire relation has been scanned. The tuple-at-a-time algorithms for each of the relational operations under consideration are detailed here.

Algorithm 1 Tuple-at-a-time selection

```
1: procedure SELECT(child, p)          ▷ Child operator, filtering predicate
2:     do
3:         t ← NEXT(child)
4:     while t ≠ nil and p excludes t
5:     return t
6: end procedure
```

Algorithm 2 Tuple-at-a-time projection

```
1: procedure PROJECT(child, exprs)          ▷ Child operator, projections
2:     t_old ← NEXT(child)
3:     if t_old = nil then return nil
4:     end if
5:     t_new ← NEWTUPLE()
6:     for each e ∈ exprs do
7:         ADDATTRIBUTE(t_new, t_old, e)
8:     end for
9:     return t_new
10: end procedure
```

Algorithm 3 Tuple-at-a-time nested-tuple join

```
1: procedure NTJoin(left, right, p, t_left)
2:     while true do
3:         t_right ← Next(right)
4:         if t_right = nil then
5:             t_left ← Next(left)
6:             if t_left = nil then
7:                 return nil
8:             end if
9:             Rewind(right)
10:            t_right ← Next(right)
11:        end if
12:        if Joins?(t_left, t_right, p) then        ▷ If tuples join for predicate
13:            return Join(t_left, t_right)
14:        end if
15:    end while
16: end procedure
```

These algorithms all assume no order within the relation. They are memory efficient in the sense that they require little more memory than that of a single tuple. However, these algorithms potentially require more time than needed assuming some order on the input relations. For instance, the join algorithm presented is the classic nested-tuple join, which loops through each tuple in the left input for each tuple in the right input. If the order of one of the relations is known for the join predicate, one can process each tuple of the unordered relation exactly once by logarithmically searching for the first tuple matching the join criterion and iterating over the rest. This reduces the computational and I/O complexity cost from the product of the sizes of the two input relations to $|U|\log_2(|O|)$[4] on average where U is the unordered relation and O is the ordered relation. Other relational algorithms similarly benefit from the use of indexes to reduce the total number of tuples processed.

[4]In the degenerate case where every tuple from the unordered relation matches every other tuple in the ordered relation, the worst-case complexity is still quadratic.

Algorithm 4 Tuple-at-a-time one-sided index join

```
1: procedure OSIJoin(indexed, unindexed, p, t_unidx, leftindexed)
2:     while true do
3:         t_idx ← Next(indexed)
4:         while t_idx = nil do
5:             t_unidx ← Next(unindexed)
6:             if t_unidx = nil then
7:                 return nil
8:             end if
9:             t_idx ← FindFirstMatching(indexed)
10:        end while
11:        if Joins?(t_idx, t_unidx, p) then
12:            if leftindexed then
13:                return Join(t_idx, t_unidx)
14:            else
15:                return Join(t_unidx, t_idx)
16:            end if
17:        end if
18:    end while
19: end procedure
```

2.4 Previous Work

Since most embedded systems use flash memories, many people have put effort into creating safe, efficient algorithms for managing data on such devices. Gal et. al. [GT05] provide key algorithms including those for block-mapping and erase-unit reclamation with wear levelling. They then go on to describe the implementation details of several file systems for flash memories. Indexing structures for the two most common forms of flash memory, NOR flash and NAND flash, have also been the focus of many authors ([LZYK+06], [KJKK07], [AGS+09]). Sorting algorithms for flash memories have also been devised [CL10].

COUGAR [BGS01] and TinyDB [MFHH05] are data management systems for sensor networks. Query translation and parsing is performed off-device. The databases read sensors as if they were relations distributed across a network of devices and a single lead node coordinates queries. Queries involving selections, projections, joins, aggregation, and grouping are supported through an SQL-like interface (TinyDB) or XML (COUGAR). TinyDB additionally allows users to query sensor readings directly at fixed

sampling periods. The sampling periods can be directly specified, or alternatively, a lifetime goal in hours, days, or joules used by the system to estimate appropriate sampling rates can be provided. Sensors can be optionally read based on their voltage requirements. COUGAR also provides mechanisms for detailing the intended duration of a query.

PicoDBMS [PBVB01] [ABP03] implements a relational DBMSs for small computing devices without on-board query translation. PicoDBMS provides a design framework that recomputes partial query results as much as possible to reduce memory requirements. Further, PicoDBMS implements this framework to demonstrate the relationship between memory requirements and execution time. The algorithms described within the work [ABP03] adapt to available memory. Slow execution time results from limited memory available. As the available memory grows, execution time falls drastically. PicoDBMS was specifically designed for smart cards where security is of great importance. A key application is in personal health cards that track patient records [PBVB01].

DBMSs for less-constrained embedded devices such as smart phones currently exist. The popular SQLite[5] is used on iOS and Android phones and tablets all the way up to web browsers on desktop computers. Other relational databases, including H2 Database Engine[6], SQL Server Compact[7], and MiniSQL[8] also target less-constrained embedded systems. Non-relational databases for similar computing devices include UnQLite[9] and BerkeleyDB[10]. None of these systems meet the extreme code space and memory restrictions of the smallest of devices.

Antelope [TD11] is a relational database for constrained devices that leverages AQL, a language similar to but not compliant with SQL. AQL has separate statements for declaring a relation and adding attributes to a relation. Antelope requires indexes for joins and aggregation. It is also able to use indexes to speed up projections and selections. Further, Antelope requires a fixed amount of RAM totalling over 3.7KB. For example, the SELECT expression `2*x/3 + 7*y` defining a new projected attribute is not supported in Antelope. Expression evaluation is supported via a shunting-yard style algorithm for parsing to a post-fix byte code to be evaluated.

[5] http://www.sqlite.org/
[6] http://www.h2database.com/
[7] http://www.microsoft.com/en-us/sqlserver/
[8] http://www.hughes.com.au/products/msql
[9] http://www.unqlite.org/
[10] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

Chapter 3

LittleD Implementation Summary

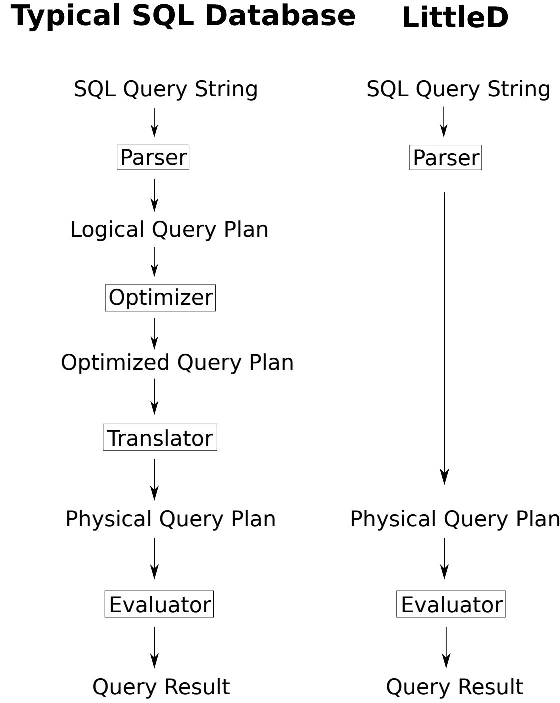
LittleD uses tuple-at-a-time algorithms to keep memory usage to a minimum. For larger datasets involving joins, simple indexes are used to achieve reduced computational and I/O complexities. LittleD specifically supports in-order indexes, also called in-place indexes, where the desired order of the tuples is assumed to be the physical order of the tuples in the relation. These techniques are not new but represent a good basis for the execution of queries.

The major difference between LittleD and other systems is the query translation procedure. Traditional query parsing and optimization is memory and code space intensive. An auto-generated lexical analyzer, also known as a lexer, cannot be used since the resulting binary is too large for the constrained space of the target devices. Using an auto-generated parser is likewise impossible. Further, traditional parsing techniques involve the use of in-memory parse-trees. This extra demand on RAM consumption means that queries whose executable form could potentially fit in the available memory may become unanswerable. As such, the lexer and parser were hand written to directly translate SQL into executable byte-code. The translation process involves parsing, validation, and optimization all in a single component.

Table 3.1: 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 expressions	Index-based

Figure 3.1: Query Processor Architecture for a typical SQL database and LittleD



To achieve a single translator component, the parsing order of the clauses for `SELECT`-statements is fixed. The first clause to be parsed is the `FROM` clause so that attribute validation can be performed on-the-fly. Attribute validation ensures specified attributes throughout the query are neither ambiguous (`SELECT a FROM A, B WHERE A.a > B.a`) nor non-existent. Once the `FROM` clause has been parsed (and the scans and joins defined there are created), the `WHERE` clause is parsed. Finally, the expressions within the SELECT clause are parsed and validated.

Despite the lack of a distinct query optimizer, some optimization involving joins is possible. Since join operators consume a considerable amount of memory, re-ordering joins is not possible. To achieve relatively ideal but not necessarily optimal join ordering, the following transformations are used:

– Un-indexed joins are converted to indexed joins whenever possible. The filtering predicate must be compatible with an available index for one of the join's input relations. An index is compatible with a predicate

the if index sorts the relation using the attributes from the relation in the same way they are used in the joining predicate. For instance, a relation's index which orders on attribute x would not be compatible with the join predicate `x % 7 = y` but would be compatible with the join predicate `x = y`.

– Left-deep join trees are used exclusively. Standard join algorithms iterate over each tuple in the right relation multiple times for each tuple in the left relation. If the input relation on the right is itself a join, that join must be re-computed multiple times. If it is on the left, it must be computed only a single time. Since indexed joins can have either the left or the right relation as the ordered relation, left-deep trees generally provide good performance characteristics. Left-deep trees are also easily constructed. During `FROM` clause parsing, each relation can be initialized in the order they appear. Then the joins are initialized in forward order. That is, after scans for each relation in `FROM` R_1, R_2, ..., R_n have been initialized, memory is allocated for the $n-1$ join operators. Then the first join J_1 between R_1 and R_2 is initialized. The second join between J_2 and R_3 is initialized, and so forth until all the joins have been constructed.

A key requirement for this type of database is to use a memory allocator that is as compact as possible. To accomplish this task, an allocator was written using two stacks. An array of fixed sizes is declared statically and then used as the allocation resource. The beginning and end of the array are initially the tops of the *front stack* and *back stack*, respectively. New allocations always occur on the top of one of these stacks, but de-allocations may occur in any order. The allocator is able to detect out-of-order de-allocations and mark these regions such that once the top of the stack is finally freed, any previously marked regions immediately under the top are also de-allocated. Due to the natural flow of data in relational databases, a fixed order of allocations and de-allocations is common.

There are further advantages to using such a memory allocator. Until the first de-allocation, no memory can possibly be fragmented. Since in almost all cases the de-allocation of memory occurs all at the end, there is effectively no memory fragmentation. The cost of allocating a new segment on the front of the stack is two pointers and on the back the cost of allocation is one pointer. Detecting out-of-memory errors is accomplished by simply checking if a new allocation will cause the tops of the stacks to collide. By knowing ahead of time how much memory is required to execute an execution plan, out-of-memory problems can be detected well before runtime. Valuable

energy is thus conserved and segmentation faults are also avoided, which could lead to potential data corruption and other destructive behavior. Many of the algorithms used to convert SQL into an executable plan require the use of one or two stacks. In converting mixed-infix expressions into postfix only byte code, the front stack is used for the resulting expression while the back stack is used to determine operator order based-on operator precedence and parentheses. Using the front stack for the resulting evaluable expression means not having to reverse the expression once complete.

To reduce overhead even more, special functions were implemented which allows for the top most memory sections on each stack to be extended in size without modifying the current data. Not only does this reduce memory overhead, it also significantly simplifies translation efforts. For example, during the parsing of expressions, the size of byte-code representation of an expression does not need to be known ahead of time, reducing the amount of work needed for overall execution. Each time a new part of the expression must be added, the stack is simply resized.

Chapter 4

Experimental Results and Discussion

4.1 Experimental Setup

In order to compare LittleD's performance characteristics with Antelope, a common execution platform was needed. A Zolertia Z1 device simulated by the MSPSim emulator served precisely this purpose. MSPSim is cycle-accurate. Both databases were compiled for the Contiki real-time operating system [Con] and used the Coffee File System [TDHV09] to store relation data.

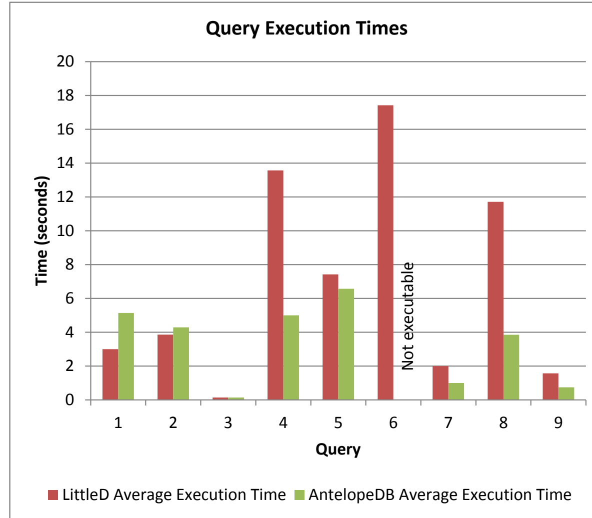
The executed queries as expressed in both AQL and SQL are in Table 4.2. The input relation's size and output relation size is also noted there. Execution times are given in Figure 4.1 while memory usage is given in Figure 4.2. Further, the *Resource Product* is defined to be the product of the amount of memory used during a query multiplied by the number of seconds taken to execute a query. This metric is meaningful in that it describes how well the database balanced execution time, which is directly proportional to the number of clock-cycles used and thus the amount of energy consumed, and the memory required.

Both databases leverage inline indexes on the first attribute for all tables and prefer indexing the right relation within a join whenever possible. For LittleD, this reduces the memory needed when upgrading un-indexed joins to indexed joins. As noted before, LittleD is capable of executing completely un-indexed joins, while Antelope is not. Benchmarks are only given for indexed joins.

4.2 Code Space and Memory Requirements

Both systems require a considerable amount of code space. As tested, including a small amount of utility code for generating relations, Antelope required 48 KB and LittleD required 55 KB. The amount of utility code

Figure 4.1: Query Execution Experimental Results



would be comparable to that needed to collect data in a real use-case.

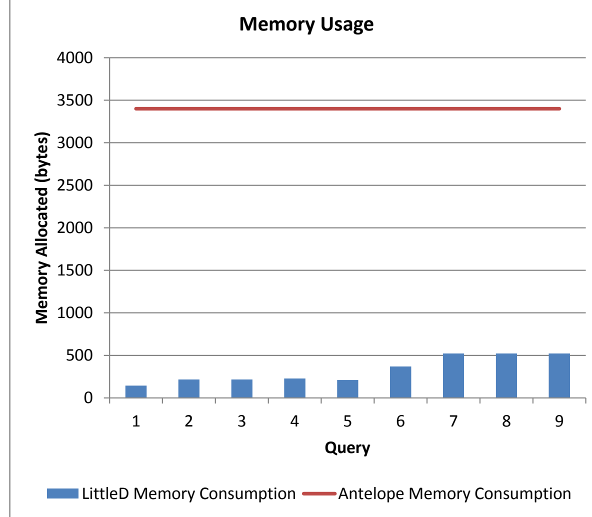
Further, Antelope requires a fixed static allocation of 3.4 KB of RAM plus between 0 to 328 KB of dynamically allocated memory [TD11]. In comparison, LittleD uses a fixed amount memory. No more than 1000 bytes of memory were required for each of the executed queries. Assuming Antelope does not allocate any dynamic memory, Antelope requires over 500% more memory than LittleD for the most memory-intensive queries. For the least memory intensive queries, Antelope requires over 1300% more memory.

4.3 Discussion of Results

The first query is a serialized scan over a large relation without any filtering predicate. LittleD does not build a projection when it is not necessary, meaning it can execute the query faster than Antelope. Antelope does not support an equivalent to `SELECT *` syntax and thus must build and execute a projection, requiring valuable execution cycles.

Queries 2 and 3 have similar execution times for both databases. LittleD and Antelope both use indexes to significantly decrease the number of tuples handled. Antelope is able to use an index again for query 4, whereas LittleD cannot due to more limited indexing support. Antelope uses an inference engine to determine from a predicate the range of values that must be iterated over. LittleD supports range expressions of the form `attr`

Figure 4.2: LittleD vs. Antelope Memory Usage

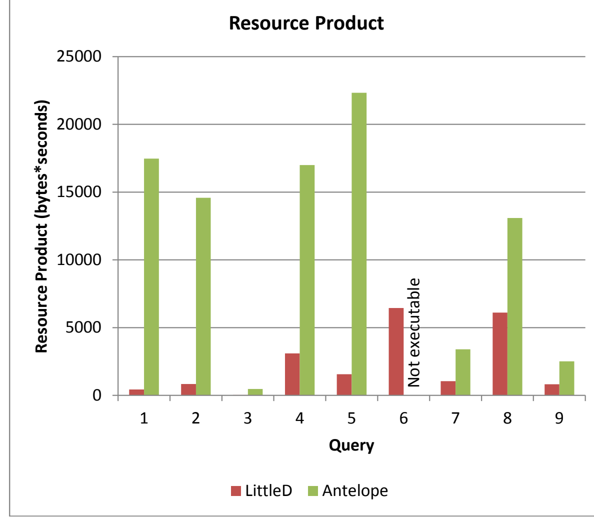


<RELOP> <INTVAL> where <RELOP> is one of =, ! =, >, <, >=, or <= and <INTVAL> is an integer value. Once an index cannot be used, as in query 5, again the two databases perform nearly equally.

Query 6 clearly demonstrates that the time taken to execute a query is linearly proportional to the number of calculations that must be evaluated per tuple. Calculating three operations over each tuple significantly increases the amount of time needed, despite the small size of the output relation. This query is not executable for Antelope because unlike LittleD, it does not have support for arbitrary projecting expressions. Antelope only supports the projection of individual attributes. In a real application, this would mean that each tuple would need to be iterated over to calculate a new value. Not only does this add considerable complexity to an application, it possibly increases the amount of memory required for a task.

Queries 7, 8, and 9 all demonstrate Antelope's superiority in terms of execution time for similar joins. However, Antelope makes simplifying assumptions, which can likely explain its speed advantage. AQL only allows for equijoins, whereas LittleD supports arbitrary inner joins via SQL. Furthermore, experimentation revealed that the implementation of the join algorithm did not always return correct results. The benchmarks provided all returned correct output relations, but for a variety of other cases, the output relation was incorrect. These limitations make it difficult to compare the join performances in a meaningful way, but it should be noted that LittleD

Figure 4.3: Query Resource Efficiency Results



was never more than three times slower than Antelope for the benchmarked joins. Query 8 was slower than query 7 for both databases since a greater number of binary searches are needed to compute the results.

For all queries, the resource product for LittleD is smaller than for Antelope. In the closest case, Antelope has a resource product that is only twice as large as LittleD. In the worst case for Antelope, its resource product is over 40 times larger than that of LittleD.

Table 4.1: Queries Executed

Query	LittleD SQL String	Antelope AQL String
1	`SELECT * FROM r`	`SELECT * FROM r`
2	`SELECT * FROM r WHERE attr0 > 4999`	`SELECT attr0, attr1 FROM r WHERE attr0 > 4999`
3	`SELECT * FROM r WHERE attr0 < 100`	`SELECT attr0, attr1 FROM r WHERE attr0 < 100`
4	`SELECT * FROM r WHERE attr0 >= 7 AND attr0 <= 6009`	`SELECT attr0, attr1 FROM r WHERE attr0 >= 7 AND attr0 <= 6009`
5	`SELECT * FROM r WHERE attr1 < 100`	`SELECT attr0, attr1 FROM r WHERE attr1 < 100`
6	`SELECT attr0, attr2/4, (attr1+1)*2 FROM r WHERE attr1 < 10 OR (attr2 / 4) = 13`	*No Executable Equivalent*
7	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`
8	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`
9	`SELECT * FROM l, r WHERE l.attr0 = r.attr0`	`JOIN l, r ON attr0 PROJECT attr0, attr1`

Table 4.2: Information about queried relations. For joins, table l has same schema as table r.

Query	Relation Schema	Relation Sizes	Output Tuples
1	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	10000
2	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	4999
3	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	5000
3	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	100
4	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	6003
5	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	6003
6	`CREATE TABLE r (attr0 INT, attr1 INT, attr2 INT)`	$\lvert r\rvert = 10000$	6
7	`CREATE TABLE r (attr0 INT, attr1 INT)`	$\lvert l\rvert = 100$, $\lvert r\rvert = 1000$	100
8	`CREATE TABLE r (attr0 INT, attr1 INT)`	$\lvert l\rvert = 1000$, $\lvert r\rvert = 100$	100
9	`CREATE TABLE r (attr0 INT, attr1 INT)`	$\lvert l\rvert = 100$, $\lvert r\rvert = 100$	100

Chapter 5

Conclusion

In this work, LittleD, a relational database for sensor nodes and embedded computing devices, was presented. A tuple-at-a-time approach to query algorithms was used. The novel features of LittleD include a single-component SQL to executable-plan translator and a memory allocator using two stacks within a single memory region.

Experimentally, LittleD is always more memory efficient than Antelope, the database that most closely matches the goals of LittleD, while having comparable execution time performance and similar code space requirements. No query for LittleD was more than three times slower than a comparable query for Antelope, though Antelope always used at least five times more memory than LittleD. LittleD supports the more familiar SQL standard including more general expressions in projections in comparison with Antelope.

This work also provides useful insights to applications of database technology with microprocessors. The use of indexes drastically improves the performance of many queries and should thus be regarded as a key component of any database targeting resource constrained platforms. The resources available, namely memory, stable storage, code space (ROM), and execution time and energy, are not equally important. While memory is limited, LittleD has low memory requirements for reasonable `SELECT-FROM-WHERE` queries. If a database system cannot fit into the code space of a device, it can never be used, and thus it should be considered at present the most constraining resource. Almost half of the ROM required for LittleD to execute is consumed by the query translation system. While SQL translation is possible on device, it is not necessarily beneficial for all applications. Finding ways to reduce the complexity of the parser, possibly by moving query translation off device, would allow for smarter algorithms to be used in query execution, possibly saving execution cycles and thus energy. Device manufacturer's would be wise to give these devices more ROM to accommodate for more sophisticated solutions. Though not tested here, all resources become significantly more taxed when sorting and aggregation are also included. Developers using LittleD or similar systems may well have to choose which features to compile for an application due to such resource limitations.

5.1 Future Work

Future research will focus on further reducing resource costs. A more sophisticated indexing strategy will be investigated for data where an order cannot be assumed at insertion time. Finding ways to better leverage the indexes throughout the query execution engine will allow for lower RAM and energy usage. Potentially moving query translation off-device could also save valuable code space. Improved sorting methods such as MinSort [CL10] will also be implemented.

Bibliography

[ABP03] Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. Memory Requirements for Query Execution in Highly Constrained Devices. VLDB '03, pages 694–705. VLDB Endowment, 2003. → pages 2, 14, 18

[AGS+09] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, August 2009. → pages 17

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. MDM '01, pages 3–14, London, UK, 2001. Springer-Verlag. → pages 17

[CL10] Tyler Cossentine and Ramon Lawrence. Fast Sorting on Flash Memory Sensor Nodes. IDEAS '10, pages 105–113, New York, NY, USA, 2010. ACM. → pages 17, 30

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970. → pages 3, 9

[Con] Contiki Operating System. `http://www.contiki-os.org/`. Accessed: 2013-08-15. → pages 23

[DKO+84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. *SIGMOD Rec.*, 14(2):1–8, June 1984. → pages 14

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660, September 2013. → pages 1

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993. → pages 13

[GT05] Eran Gal and Sivan Toledo. Algorithms and Data Sructures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005. → pages 2, 17

[KJKK07] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: An Ordered Index Structure for NAND Flash Memory. EMSOFT '07, pages 144–153, New York, NY, USA, 2007. ACM. → pages 17

[LZYK+06] Song Lin, Demetrios Zeinalipour-Yazti, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Efficient Indexing Data Structures for Flash-Based Sensor Devices. *Trans. Storage*, 2(4):468–503, November 2006. → pages 17

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. → pages 17

[PBVB01] Philippe Pucheral, Luc Bouganim, Patrick Valduriez, and Christophe Bobineau. PicoDBMS: Scaling Down Database Techniques for the Smartcard. *The VLDB Journal*, 10(2-3):120–132, September 2001. → pages 18

[TD11] Nicolas Tsiftes and Adam Dunkels. A Database in Every Sensor. SenSys '11, pages 316–332, New York, NY, USA, 2011. ACM. → pages 2, 18, 24

[TDHV09] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. IPSN '09, pages 349–360, Washington, DC, USA, 2009. IEEE Computer Society. → pages 23