Developing the SQL IonDB Query Language

by Dana Klamut

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

B.A. COMPUTER SCIENCE HONOURS

in

Irving K. Barber School of Arts and Sciences

(Computer Science)

Supervisor: Dr. Ramon Lawrence

THE UNIVERSITY OF BRITISH COLUMBIA
(Okanagan)

April 2018

Abstract – The Internet of Things (IoT) continues to engage society as its capabilities extend. With these IoT devices aiding daily life, the data created through their use grows rapidly. As these devices are memory constrained, IonDB was created to serve as a key-value database able to accommodate these constraints, in a performance competitive manner. However, the methods required for querying the data are IonDB-specific, and therefore may not be intuitive to users. Professionals use the SQL query language, which is a well-known, standard query language. Due to the prevalence of IonDB's users' familiarity with the structure of SQL, this document demonstrates the creation of an SQL IonDB query language, known as IINQ. The implementation and format of this query language, which adheres to IonDB's standards of speed and memory, are presented in detail. Experimental results and testing was performed on the popular Arduino system.

Table of Contents

Table of Figures

Table of Listings

1. Introduction

In the age of the Internet of Things, microcontrollers and embedded processors are seamlessly integrated into everyday life. As technology is continuously assimilated into society at increasing rates, these devices are gathering and storing seemingly endless amounts of data. This data has the capacity to drive decision-making. The need for accessing the data in meaningful ways led to the creation of query languages for users to access information in a database based on their particular request, without the necessity for the user to have extensive knowledge of the database configuration [1]. SQL, originally known as Structured English Query Language, or SEQUEL, is the standard query language for relational database management systems [2]. In addition, many professions continue to require SQL skills including data analysts [3]. As SQL is universally well-known as a standard query language, it can be extended to new areas, even as database technology evolves.

With the active presence of embedded devices, the opportunity arises to extend their functionality to users. This includes both data storage and the possibility for further manipulation. The action of data being processed locally on these memory-constrained devices has been the main focus of IonDB, a key-value database designed specifically for embedded devices [4]. With IonDB's already established foundation, an Ion Integrated Query (IINQ) language can be built upon its basis. The name IINQ was chosen in keeping with the "Ion" prefix of IonDB, and as a tribute to LINQ, Microsoft's .NET Language Integrated Query [5]. After presenting a more thorough background on embedded devices, IonDB, and query languages, the motivations of this project will be discussed. To follow, other research with respect to databases designed for embedded devices and the framework and history of SQL is examined in detail. This section also highlights several key features of SQL and a compares SQL to other existing query languages. The implementation of each feature of IINQ is discussed in detail with respect to both user code and IINQ translated code. Thereafter, the experimental results of IINQ testing are examined, followed by final conclusions and opportunities for future work.

1.1 Embedded Devices

The Internet of Things (IoT) consists of devices that are interconnected on a network with the goal of improving and aiding society through data collection and analysis. The framework in which the IoT is built supports data being interchanged over a network between users and electronic devices [6]. It is important to note that the IoT may be decomposed into three main components: software, hardware, and communication infrastructure [6]. IINQ will focus on software designed to support the communication and manipulation of data collected by these devices.

Many of these IoT devices include embedded devices, including one particular brand known as Arduino, which can be programmed for communication and operations using internet connectivity and various sensors [7]. The Arduino was created in 2005 by Massimo Banzi and David Cuartielles [8]. Arduinos are 8-bit microcontroller AVR devices that are designed to be inexpensive, self-contained, and easy-to-use, including an IDE for programming [8]. Due to these unique characteristics which are suited for real-world monitoring and their capabilities for internet access, Arduinos can easily be configured to function as IoT devices. Embedded devices, including Arduinos, are often superior in these applications as they allow for the processing of the generated data to occur directly on the devices instead of transferring the data [9].

1.2 IonDB

IonDB has been actively developed since the summer of 2014. It is as a key value database for embedded devices, which is easy to use and performance competitive [4]. IonDB includes six unique data structure implementations including: skiplist, open address hash, open address file hash, B+ tree, flat file, and linear hash [10] [11]. Several of IonDB's most notable advancements since its first publication at IEEE (CCECE) in 2015 include the implementation of two supplementary data structures, as well as a C++ language wrapper for increased ease of use of the Arduino API. Listing 1 shows the original Arduino API usage to create a file-based dictionary. In contrast, Listing 2 exemplifies the creation of a file based dictionary using the C++ wrapper.

```
// Declare the dictionary and handler structs
ion_dictionary_handler_t    handler;
ion_dictionary_t            dictionary;          // Initialize handler
ffdict_init(&handler);
dictionary_create(&handler,
                  &dictionary,
                  -1,
                  key_type_numeric_signed,
                  sizeof(int),
                  60,
                  10);
```
Listing 1. IonDB Dictionary level interface dictionary creation

```
Dictionary<int, int> *dictionary = new FlatFile<int, int>
                                          (-1,
                                          key_type_numeric_signed,
                                          sizeof(int),
                                          sizeof(int),
                                          7);
```
Listing 2. IonDB C++ Wrapper dictionary creation

As shown in Listing 1, the `init` function is required to affix the dictionary-type specific functions to the API to ensure the instance the dictionary adheres to the data structure implementation [9]. Listing 2 in comparison to Listing 1 illustrates the simplicity of the C++ wrapper as well as its conveniently minimalistic user code.

IonDB's other most notable recent accomplishment is a second publication at IEEE (CCECE) in 2017. This publication features the continuous integration platform for Arduino embedded software, as used by IonDB [12]. With the use of this platform, IonDB's development is continuously integrated, including strict testing, ensuring that the codebase is maintained and progressed adhering to a high quality of work. IonDB has progressed as a fully functional database, including many choices of unique data structures to be selected with respected to user preferences and constraints. With its continued development, and loyal following on GitHub, IonDB provides a structured basis for further growth.

The dictionary level interface of IonDB is written in the C programming language, as is IINQ in this document, unless otherwise explicitly stated as Java or C++ code. As a key-value database, the main functionality of IonDB includes insert, update, and delete operations for records, accessible through both the dictionary level interface and the C++ wrapper. Listings 3-4 below demonstrate the user code necessary, through use of the dictionary level interface or the C++ user interface, respectively, to perform an insert operation.

```
// Instantiate the key and value to be inserted
ion_key_t   keys             = IONIZE(42, int);
ion_value_t value            = (ion_value_t) "Hello IonDB";

// Insert record into dictionary
dictionary_insert(&dictionary, key, value);
```

Listing 3. IonDB dictionary level interface Insert operation

```
// Instantiate value to be inserted (may omit if value is a base type)
ion_value_t value            = (ion_value_t) "Hello IonDB";

// Insert record into dictionary
dictionary->insert(42, value);
```

Listing 4. IonDB C++ user interface Insert operation

Listing 4 demonstrates how IonDB's user interface has evolved from that of Listing 3 to be more simplistic and require less code and IonDB specific knowledge from users.

The current query capabilities of IonDB include all records, range, and equality cursor queries. An equality query produces resultant records from within the specified dictionary with a matching key to the key given by the user and cannot produce records corresponding to a value field in the database. The dictionary level interface code and the C++ user interface code required to perform an example all records query using IonDB are shown in Listings 5 and 6, respectively.

```
ion_predicate_t predicate;
dictionary_build_predicate(&predicate,
                           predicate_equality,
                           IONIZE(10, int));

ion_dict_cursor_t *cursor = NULL;
dictionary_find(&dictionary, &predicate, &cursor);
ion_record_t ion_record;
ion_record.key        = malloc(key_size);
ion_record.value      = malloc(value_size);

ion_cursor_status_t cursor_status;

while ((cursor_status = cursor->next(cursor, &ion_record)) ==
cs_cursor_active || cursor_status == cs_cursor_initialized) {
      printf("\tKey: %d, Value: %s\n", NEUTRALIZE(ion_record.key, int),
      (char *) ion_record.value);
}
```

Listing 5. IonDB dictionary level interface All Records query

```
Cursor <int, int> *cursor = dictionary->equality(10);

while (cursor->next()) {
      int key     = cursor->getKey();
      ion_value_t = cursor->getValue();
}
```
Listing 6. IonDB C++ user interface All Records query

As shown in the listings thus far, it is evident that IonDB has undergone improvements with respect to the user code required to perform operations. However, it still has its own API and does not follow a standard such as SQL.

1.3 Query Languages

Familiarity with databases, including the manipulation and retrieval of data from them using a query language, is a key component of any computer science education [3]. Queries are defined as requests to databases, with the structure in which these queries are written denoted as a query language [1]. SQL was one query language intended for bridging the gap between the data stored in databases and those who wished to access it, without using additional software or enlisting a programmer [1].

In 1986 the first edition of the SQL query language was released, as developed by the American National Standards Institute [13]. Learning to compose and execute SQL continues to be a main focus in many computer and data science courses and remains applicable to numerous professions [3].

Other technologies developed for the storage and manipulation of data include NoSQL systems, which abbreviates “No SQL” or “Not only SQL” [14]. Unlike databases such as IonDB tailored to memory constrained environments, NoSQL systems are capable of handling big data, including storage and processing [14].

1.4 Motivations

With the advances of IonDB, more features have been made possible. As IonDB has developed into a fully functional key-value database, its areas of expansion have also grown. One area of interest in particular, is that of an SQL query language to complement IonDB, as the current fundamental operations supported by IonDB such as create table, insert, update, and delete may not be instinctive to even experienced programmers. Due to the prevalence of use of SQL as a query language, IonDB queries adhering to the standard of SQL would be widely understood by the majority of users.

As IonDB is already fully functional, one of the main challenges of IINQ is encapsulating this functionality within an SQL user format while adhering to the performance and memory standards set by IonDB. With doing so, the user code necessary for accomplishing these operations must also be kept minimal and simplistic, including all underlying dictionary level C code being concealed from the user.

IonDB is the perfect base for an SQL query language as it is a key value database, hence making it recognizable to users experienced with any key-centric data model, such as semantic models, object models, XML, or RDF [15]. The implementation of a universally understood SQL query language for IonDB will extend both IonDB's functionality and user following.

1.5 Contributions

The contribution of this thesis is the development and performance evaluation of an SQL interface for IonDB. Since embedded devices have insufficient memory and CPU resources to process SQL queries on device, the key contribution is a compile-time optimization and automatic code generation that translates user-specified SQL queries into optimized C code for execution on the device. The typical steps of query optimization and processing done by a relational database are performed offline during compilation which dramatically improves performance and ensures that the SQL-enabled IonDB implementation can run on very resource-limited devices.

2. Background

2.1 IoT Databases

IonDB was created to bridge the gap of storing generous amounts of key-value data within a memory limited environment, however, other work has also been done in this area.

Other research includes a "small-footprint" database designed for Windows CE, a Microsoft embedded device and mobile operating system [16]. This database in particular supports relational databases through the use of SQL Server, using both built-in SQL functionality and embedded environment specific features [16]. Similar to IINQ, SQL Server for Windows CE understands the importance of adhering to SQL syntax standards to accommodate a wide audience of users [16]. Another memory-conscious database that has been designed focuses on using flash memory for data storage and manipulation [17].

In addition, FAME-DBMS was also created with these constraints in mind [18]. However, FAME-DBMS remains unique as in addition to being suited for embedded systems, this database management system (DBMS) uses the *software product line* to tailor data management to a variety of scenarios [18]. This work was in part motivated by research on Berkley DB, an embedded and server systems embedded DBMS [18] [19]. As Berkley DB is not designed solely for use on embedded systems, some configurations must be made by the user to alter its original format of 96,000 lines of code in order to fit on memory constrained devices, making it less ideal for embedded systems users [19].

Although the advancements of databases designed for memory limited environments continue, they are not necessarily a novel area of research. In 1999 IBM began producing small system databases, coined as their *DB2 database family* [20]. This announcement by IBM followed Oracle releasing the *Oracle8i Lite,* another system created to address the need for small storage on IoT devices [20].

2.2 SQL

Following the acceptance of Dr. E. F. Codd's relational model and relational database management systems, SEQUEL was created by IBM Corporation, Inc., and later made commercially available by Relational Software Inc. in 1979, now known as Oracle, who rebranded the query language as SQL [2]. SQL serves as an intermediary tool which non-technical users can master, without being experienced programmers, to access the data in relational databases [1]. Since its development, SQL has become widely known and used, ranking as the greatest in-demand data language in 2012 [21]. This is exemplified by relational database technology being valued at 40 billion US dollars annually, compared to 3.5 billion US dollars for NoSQL systems [15]. Hence, it is clear that both systems are widespread, however, relational databases and therefore SQL technology exhibits a larger following. In addition, relational databases also impose relations between data tables and reinforce ACID properties, which are used to ensure valid data consistencies and concurrent transactions [14].

2.2.1 Create Table

SQL schema statements allow users to describe the data structures and schema to be stored and can be used to create new tables [13]. A schema statement used to create a new table in the database includes the keywords `CREATE TABLE`, followed by the desired schema information for the table [13]. An example schema statement for creating a table using SQL syntax is shown in Listing 7.

```
CREATE TABLE Dogs (
      id INT,
      name VARCHAR(30),
      breed VARCHAR(40),
      age INT,
      hunger INT,
      PRIMARY KEY(id)
);
```
Listing 7. SQL Create Table schema statement

As demonstrated above, a schema statement for creating a table includes defining the values to be stored in the table and their corresponding data types. It is also notable that the primary key of the table is also defined in this statement. The key serves as a group of fields whose values can uniquely identify a specific record in the table and should be selected carefully as to ensure that it is a distinctive identifier [15].

2.2.2 Insert

SQL also supports data manipulation statements, which include `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements [13]. Insert statements are used to populate tables created through `CREATE TABLE` statements. The necessary components of an `INSERT` include the table name in which the values are to be inserted into; the column names that data is being provided for, this is not required if data is provided for every column included in the table; and the values corresponding to those columns which will be populated into the table [13]. To comply with proper SQL syntax, the values listed must be in the same order as the table schema, or as defined in the column list, with their value types adhering to the data types defined for each column in the table's schema statement [22]. The value to be inserted corresponding to the primary key column must be unique, or else the data statement will be rejected by most SQL systems [15]. An example `INSERT` statement following SQL syntax is shown in Listing 8 and corresponds to the schema as defined in the `CREATE TABLE` statement of Listing 7.

```
INSERT INTO Dogs (id, name, breed, age, hunger)
      VALUES (1, 'Minnie', 'French Bulldog', 5, 2);
```
Listing 8. SQL Insert data statement

The result of an `INSERT` statement is that of one row being added to the database specified [13].

2.2.3 Update

Once the table has been populated, modifications to the data may be necessary. These alterations are possible through `UPDATE` SQL data statements [13]. The information necessary to complete an `UPDATE` operation is similar to `INSERT` operations, as it involves the table name, as well as the fields to be updated and their corresponding data values. Listing 9 demonstrates an example `UPDATE` statement as defined by SQL and following the schema of Listing 7.

```
UPDATE Dogs SET
      age = 4,
      hunger = 0
WHERE breed = 'Chihuahua';
```
Listing 9. SQL Update data statement

Note that a new keyword included in Listing 9 is `WHERE`. The `WHERE` clause is optional in `UPDATE`, `DELETE`, and `SELECT` statements, and acts as a filter on the dataset, limiting the rows produced by the query [23]. It does so by evaluating the conditional statement provided by the user to true or false, and then either includes the row in the data set or excludes it, respectively [23].

2.2.4 Delete

The deletion of rows from a table occurs based on the table name provided by the user and the optional `WHERE` clause. As discussed above, the result set is limited to those rows matching the `WHERE` clause condition [23]. Once a `DELETE` is performed, all rows matching the condition are erased from the table. An example `DELETE` operation is shown in Listing 10.

```
DELETE FROM Dogs
WHERE age > 10;
```
Listing 10. SQL Delete data statement

If Listing 10 were to be run on a populated table named *Dogs*, it would erase all of the rows in the table where the value for the field *age* is an integer greater than 10.

2.2.5 Select

The `SELECT` SQL is the fundamental data statement for accessing the table data in a meaningful way according to the user's needs, through the retrieval of data from single or multiple tables [23]. Following the `SELECT` keyword, the user must list the fields in which they wish to receive from the result set, or a `*` signifies that all columns will be returned [24]. In addition, a `FROM` clause must be stated which specifies the sources of the data to be retrieved [23]. A simple `SELECT` statement, including only `FROM` and `WHERE` clauses, is displayed in Listing 11.

```
SELECT name, age
FROM Dogs
WHERE hunger = 0;
```
Listing 11. SQL Select data statement

The above query will provide a result set of the values of the *ages* and *names* from all of the rows in the table *Dogs* where the value of *hunger* in that row is 0.

2.2.6 Drop Table

`DROP TABLE` is the simplest of the SQL commands. This statement is used to remove the table specified, including any data still contained within it, from the database [25]. The code to drop all information from the *Dogs* table, and delete the table itself, is shown in Listing 12.

```
DROP TABLE Dogs;
```
Listing 12. SQL Drop Table statement

Although `DELETE` may be used to delete all of the rows from within the table, the table continues to persist in memory until a `DROP TABLE` command is performed [25].

3. Architecture

IINQ's intuitive SQL interface uses an SQL translator that converts user SQL code into executable code on the device during compilation and building. This process is accomplished through several steps almost entirely concealed from the user. To perform an SQL command, the only code the user is responsible for writing is the command including their specifications, as outlined by IINQ SQL syntax, which is defined in detail in Sections 3.1 – 3.5. Once the user code is written, an SQL translator is run to generate the C code necessary to run the SQL command on IonDB. This is accomplished through

Java code parsing the user code and writing back to another C file in the user's directory only the C code for their current commands. Hence, the C file with the underlying generated code supporting the functionality of IINQ is re-generated depending on the user's commands as to limit the space it uses and increase IINQ's performance speed. The user commands are also updated by the Java code to follow the function signatures generated. Following the C code generation by the Java SQL translator, the code is now executable and can be compiled on device. This general process is outlined in Figure 1.

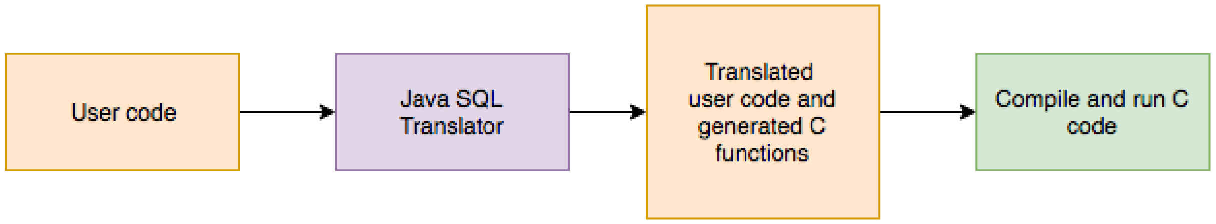


Figure 1. IINQ architecture and process flow

3.1 Create

Creating a new table in the database is fundamental to any subsequent operations. This process is accomplished through user code designed to be similar to the SQL `CREATE TABLE` command. The user code written to create a new table, as outlined in Listing 7, is shown in Listing 13.

```
SQL_execute("CREATE TABLE Dogs
      (id INT,
      name VARCHAR[30],
      breed VARCHAR(40),
      age INT,
      hunger INT,
      primary key(id));");
```
Listing 13. IINQ Create Table statement

Once the Java SQL translator is run on the user code, Listing 13 becomes translated to the code in Listing 14.

```
/*  SQL_execute("CREATE  TABLE  Dogs  (id  INT,  name  VARCHAR[30],  breed
VARCHAR(40), age INT, hunger INT, primary key(id));"); */
create_table("Dogs.inq",
            key_type_numeric_unsigned,
            sizeof(int),
            (sizeof(int) * 3)+(sizeof(char) * 70));
```
Listing 14. IINQ translated Create Table statement

The IINQ `create_table` function is fully generic and can be called repeatedly to create an infinite (within memory constraints) number of tables. The value size for the table to support, the last parameter shown in the function call of Listing 14, is calculated from the data types of each field listed in the user code and condensed for minimal computations.

3.2 Insert

IINQ has been written to follow SQL syntax as close as possible to accommodate users while still supporting the underlying functionality. The function signatures for a user to complete an `INSERT` are listed in Listing 15.

```
iinq_prepared_sql SQL_prepare (char *insert_sql);
void setParam (iinq_prepared_sql p, int field_num, void *value);
void execute (iinq_prepared_sql p);
```
Listing 15. IINQ Insert function signatures

The variation from standard SQL syntax evident in Listing 15 is the `iinq_prepared_sql` variable returned from the `INSERT` statement. This object is responsible for setting up the function pointers used for executing the command. The SQL `INSERT` command is written as a string within the `SQL_prepare()` function call. Every succeeding function call must be passed the `iinq_prepared_sql` variable, as it also stores the table identifier, as well as the key and value to be inserted. A full example of an IINQ `INSERT` statement is shown in Listing 16, following the table schema defined in Listing 13.

```
iinq_prepared_sql p = SQL_prepare("INSERT INTO Dogs
                                    VALUES (1, 'Poodle', 'Lady', 5);");
p.execute(p);
```
Listing 16. IINQ Insert statement

IINQ also supports prepared insert statements, as demonstrated in Listing 17. Prepared statements in IINQ are indicated by `(?)` within the `INSERT` SQL string. These values are then set with the `setParam` function, using the field number that the user is setting the value for (field numbers are incremented starting at 1), and the value to be set for the field. Regardless of data type, any parameterized value may be set using `setParam` as its function signature expects a void pointer type. However, this is also an area of caution for users to ensure that they are inserting the correct value type for the given column.

```
iinq_prepared_sql p = SQL_prepare("INSERT INTO Dogs
                                    VALUES (1, (?), 'Lady', 5);");
p.setParam(p, 2, "'Poodle'");
p.execute(p);
```
Listing 17. IINQ Insert prepared statement

Once the user code for the `INSERT` statement has been written, the Java code that acts as an SQL translator may be run. Listing 18 shows the resultant user C code after running the SQL translator.

```
/* iinq_prepared_sql p = SQL_prepare("INSERT INTO Dogs
                                    VALUES (1, (?), 'Lady', 5);"); */
iinq_prepared_sql p = SQL_Dogs(1, "(?)", "'Lady'", 5);

p.setParam(p, 2, "'Poodle'");
p.execute(p);
```
Listing 18. Translated IINQ Insert statement

As demonstrated above, the user's inserted statement is translated into a function call specific to the table they are inserting values into, and those values are transformed into the according parameters of the function. In addition, all of the underlying code for `SQL_Dogs`, `setParam`, and `execute` is written in a C file without any additional effort required from the user. These methods are capable of supporting multiple tables and multiple `INSERT` statements depending on the volume of commands provided by the user for that iteration of translation. All underlying code is generated to be as generic as possible for this reason, except for the table specific call that initially returns the `iinq_prepared_sql` object. The rows inserted into the table are stored indefinitely until the destruction of the table.

The IINQ functions `setParam`, `update`, and `iinq_select` all utilize the `calculateOffset(const unsigned char *table, int field_num)` and `getFieldType(const unsigned char *table, int field_num)` functions which are created upon Java translation. These functions each contain a switch statement for each table that an IINQ SQL statement is written for in the user code file. Within the switch for each table, there is another switch statement for the field number. The function `calculateOffset` returns the size of the data type specified that field number, whereas `getFieldType` returns an enum representing the data type for that specified field number. These functions allow the underlying SQL command functions to remain more generic as there is limited table-specific code.

3.3 Update

The architecture of the IINQ SQL `UPDATE` and `DELETE` commands is quite similar, and they differ from the other commands in their very altered form from standard SQL syntax following Java translation. As both functions support `WHERE` clauses, a `where` function is called by both `UPDATE` and `DELETE` for every record examined and returns a boolean value with respect to whether the record passes the `WHERE` condition. However, differing from standard SQL syntax, the user separates multiple `WHERE` clauses using commas in IINQ instead of the standard `AND` keyword. This is due to optimizing the reusable code already implemented in the Java translator, as `WHERE` is the only list in SQL commands such that the components are not comma separated. The function signature for `where` is shown in Listing 19.

```
ion_boolean_t
where( unsigned char *id, ion_record_t *record, int num_fields,
   va_list *where_clause );
```
Listing 19. IINQ Where function signature

As demonstrated in Listing 19, the parameters of `where` include the record in question, the number of items in the variable argument list, and the variable argument list of `WHERE` clauses. Within the variable argument list, we expect a field number, an operator (such as equal to, greater than, etc.), and a value for each `WHERE` clause in the `UPDATE` or `DELETE` statement. These items determine whether the record passes the `WHERE` condition and are translated from the user code.

`UPDATE` is a complex command as it contains a list of `WHERE` conditions as well as a list of updates to perform. The function signature of `update` is displayed in Listing 20.

```
void
update(int id, char *name, ion_key_type_t key_type, size_t key_size,
size_t value_size, int num_wheres, int num_update, int num, ...);
```
Listing 20. IINQ Update function signature

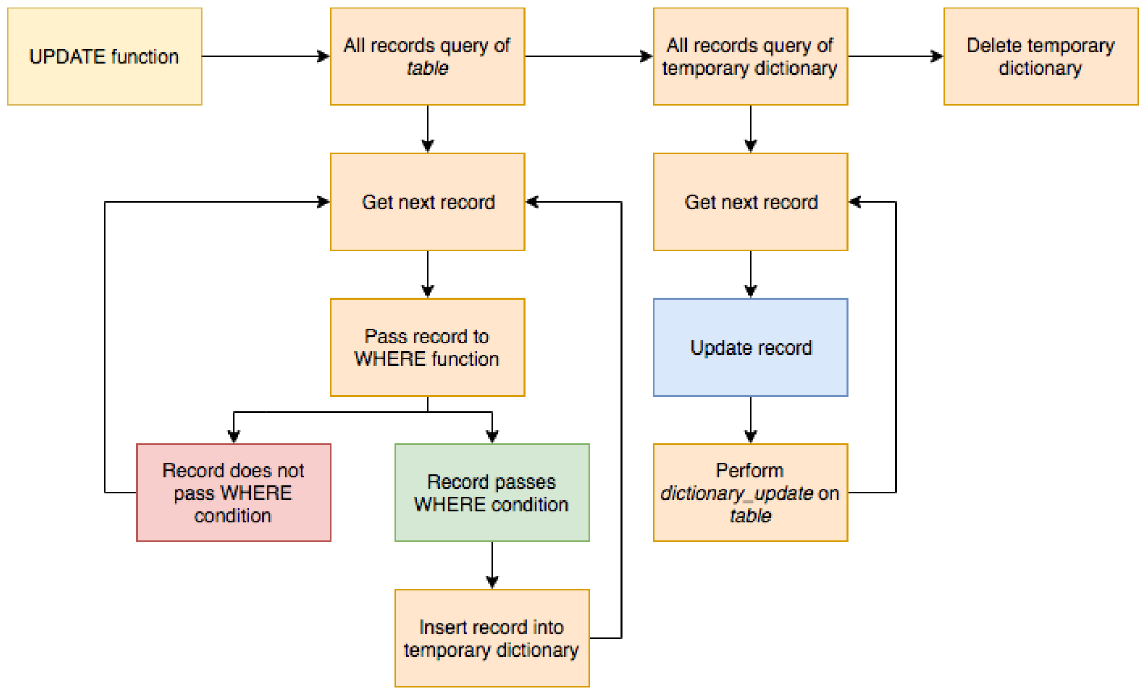
As evident in Listing 20, `update` is a variable argument function. The use of variable arguments is necessary to support a potentially infinite number of `WHERE` conditions or updates to perform. From the parameters `num_wheres` and `num_update`, the variable argument list is able to be separated correctly. Similar to `where`, the updates are translated into a field number, implicit update field number, math operator (such as add, multiply, etc.), and a value. The implicit update field number and math operator are only found in user code when the `UPDATE` includes using a field value to update a possibly different field. If the user code is simply updating a field to a new constant value, the implicit update field number and math operator are set to null. The user code and Java translated code for an implicit `update` statement are shown in Listings 21 and 22.

```
SQL_execute("UPDATE Dogs SET age = age + 5 WHERE hunger > 0;");
```
Listing 21. IINQ Update statement

```
/* SQL_execute("UPDATE Dogs SET age = age + 5 WHERE hunger > 0;"); */
update(0, "Dogs.inq", key_type_numeric_unsigned, sizeof(int),
(sizeof(int) * 3)+(sizeof(char) * 70), 3, 4, 7, 5, iinq_greater_than, 0,
4, 4, iinq_add, 5);
```
Listing 22. Translated IINQ Update statement

The translation of user code is especially necessary for `UPDATE` commands as the function call to IINQ `update` is complicated and unintuitive. However, through the use of the Java translator, the simple user code can be transformed to adhere to underlying `update` function signature without the user needing any knowledge of its structure. Therefore, the C code of `update` remains generic and can be called for any table with any variation of updates and `WHERE` conditions.

`UPDATE` operates through a variety of steps, as outlined in Figure 2.



UPDATE function
All records query of *table*
All records query of temporary dictionary
Delete temporary dictionary
Get next record
Get next record
Pass record to WHERE function
Update record
Record does not pass WHERE condition
Record passes WHERE condition
Perform *dictionary_update* on *table*
Insert record into temporary dictionary

Figure 2. IINQ Update process

As seen above, when the translated user `UPDATE` command is called, an *all records* query is performed on the table in question. Each record retrieved is passed to the `where` function. If the record passes the `WHERE` condition, it is stored in a second IonDB dictionary, to utilize the flash memory available instead of storing the records on the stack. Once all of the records in the specified table have been examined, a new *all records* query is then performed on the new dictionary storing the records to be updated. On this iteration each record is updated accordingly, and an IonDB `dictionary_update` is performed on the original table. Finally, the temporary dictionary storing the records to be updated is deleted from memory.

3.4 Delete

The `DELETE` command mimics the architecture of `update` as they both support a `WHERE` condition. The IINQ `DELETE` process is demonstrated in Figure 3.

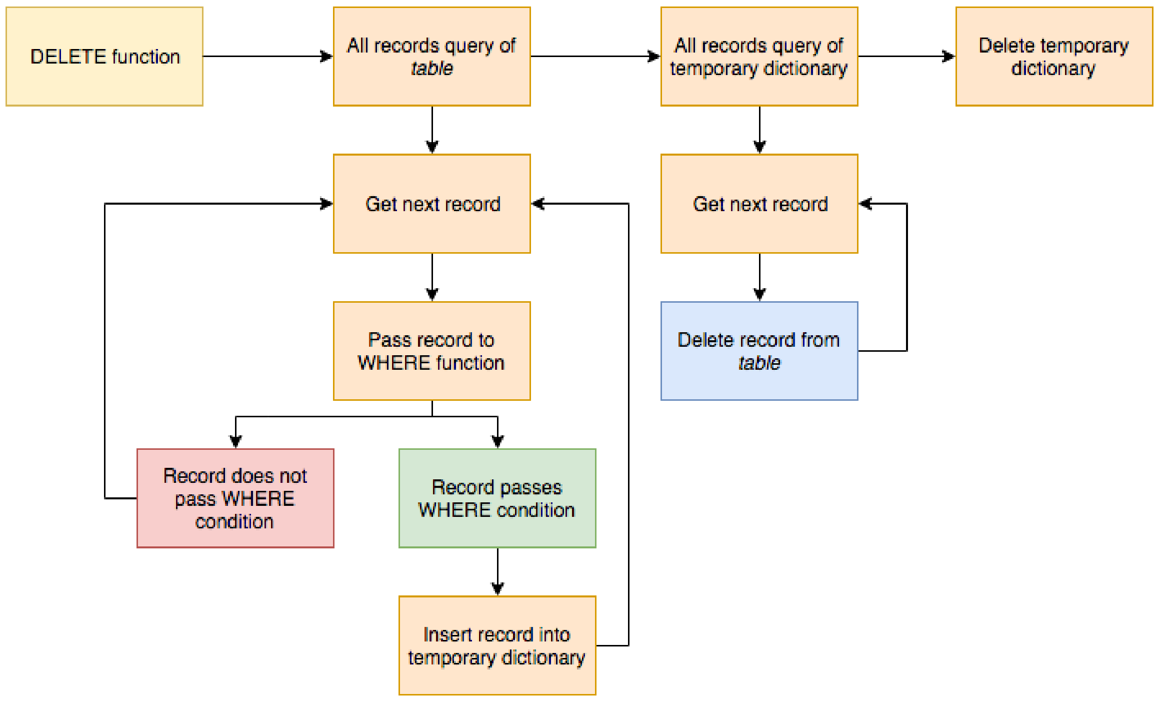

DELETE function
All records query of *table*
All records query of temporary dictionary
Delete temporary dictionary
Get next record
Get next record
Pass record to WHERE function
Delete record from *table*
Record does not pass WHERE condition
Record passes WHERE condition
Insert record into temporary dictionary

Figure 3. IINQ Delete process

The main differentiating factor between `DELETE` and `UPDATE` is the handling of the records inserted into the temporary dictionary. Unlike `UPDATE`, `DELETE` is able to insert only the keys of the records to be deleted, as verified by the `WHERE` condition, with an arbitrary integer value as the record's value instead of the true value to conserve memory. This is possible as only the record's key is necessary to perform `dictionary_delete` on the table.

The function signature for `DELETE` is shown in Listing 23 and is fully generic to support deletes on any number of tables through multiple function calls. The naming of delete has been changed to `delete_record` as `DELETE` is a reserved word in C++, therefore throwing an error when compiling in the Arduino IDE.

```
void
delete_record(int id, char *name, ion_key_type_t key_type,
size_t key_size, size_t value_size, int num_fields, ...);
```
Listing 23. IINQ Delete function signature

The user code to complete a `DELETE` command is shown in Listing 24, with the translated code to fit the `delete_record` parameters demonstrated in Listing 25. This translated code is similar to `update`, with the exclusion of a list of fields to update.

```
SQL_execute("DELETE FROM Dogs WHERE age < 5;");
```
Listing 24. IINQ Delete statement

```
/* SQL_execute("DELETE FROM Dogs WHERE age < 5;"); */
delete_record(0, "Cats.inq", key_type_numeric_unsigned, sizeof(int),
(sizeof(int) * 3)+(sizeof(char) * 70), 3, 4, iinq_less_than, 5);
```
Listing 25. Translated IINQ Delete statement

The variable arguments in a `DELETE` include the field number to check in the `WHERE` condition, the operator, and the value to compare the field value to. Through the use of the `where` function, as outlined in detail in section 3.3, `delete_record` is able to support an unlimited number of `WHERE` clauses.

3.5 Select

The `SELECT` command allows users to draw meaningful insights and conclusions from querying the records. `SELECT` can be extended to support complex queries, however, as outlined in Section 2.2.5, IINQ's implementation of `SELECT` is limited to selecting a field list from one table with an unlimited number of `WHERE` conditions. The function signature of IINQ's `SELECT` command follows standard SQL syntax, as outlined in Listing 26.

```
iinq_result_set
iinq_select(int id, char *name, ion_key_type_t key_type, size_t key_size,
size_t value_size, int num_wheres, int num_fields, int num, ...);
```
Listing 26. IINQ Select function signature

From Listing 26, it is evident that the `SELECT` function signature resembles `update` very closely. However, the parameters of `iinq_select` also include the number of fields in the field list to return, to accommodate `getInt` and `getString`. The user code to perform `SELECT`, followed by the translated `iinq_select` code, is shown in Listings 27 and 28. An `iinq_result_set` struct is returned from the `SELECT` command, modelling how an SQL `SELECT` statement is processed in Java using JDBC [26]. This struct contains a pointer to retrieve the next record stored in the temporary IonDB dictionary containing all of the records from the user specified table which passed the given `WHERE` conditions. The values of the records in the temporary dictionary are condensed to only include the fields listed in the field list.

```
iinq_result_set rs1 = SQL_select("SELECT id, name FROM Dogs WHERE
                                   age < 10;");
```
Listing 27. IINQ Select statement

```
/* iinq_result_set rs1 = SQL_select("SELECT id, name FROM Dogs WHERE age
< 10;"); */
iinq_result_set rs1 = iinq_select(0, "Cats.inq",
key_type_numeric_unsigned, sizeof(int), (sizeof(int) * 2)+(sizeof(char)
* 30), 3, 2, 5, 4, iinq_less_than, 10, 1, 2);
```
Listing 28. Translated IINQ Select statement

Following the return of the result set iterator, the user is able print or process the records accordingly. To accommodate this, `getInt` and `getString` have been implemented to allow the user to print specific fields stored within the returned record. The function signatures of these helper functions are shown in Listings 29 and 30.

```
int getInt(iinq_result_set *select, int field_num);
```
Listing 29. IINQ GetInt function signature

```
char* getString(iinq_result_set *select, int field_num);
```

Listing 30. IINQ GetString function signature

The above functions can be called with the result set to aid the user in, for example, printing a specific field value. The field number passed to the function in this case does not correspond to the field's listing in the table schema, but instead to the field's placement within the field list of the `SELECT` statement. Example user code of the user's interaction with the result set following the execution of a `SELECT` statement is displayed in Listing 31. This listing demonstrates the intentional similarity between IINQ user code and JDBC SQL statement processing.

```
while (next(&rs1)) {
   printf("ID: %i,", getInt(&rs1, 1));
   printf(" name: %s\n", getString(&rs1, 1));
}
```

Listing 31. Example IINQ Select processing

With the use of `getInt` and `getString`, the records returned by `SELECT` can be processed in any way the user wishes by accessing the fields independently, and in no particular order. An overview of the `iinq_select` process is outlined in Figure 4.

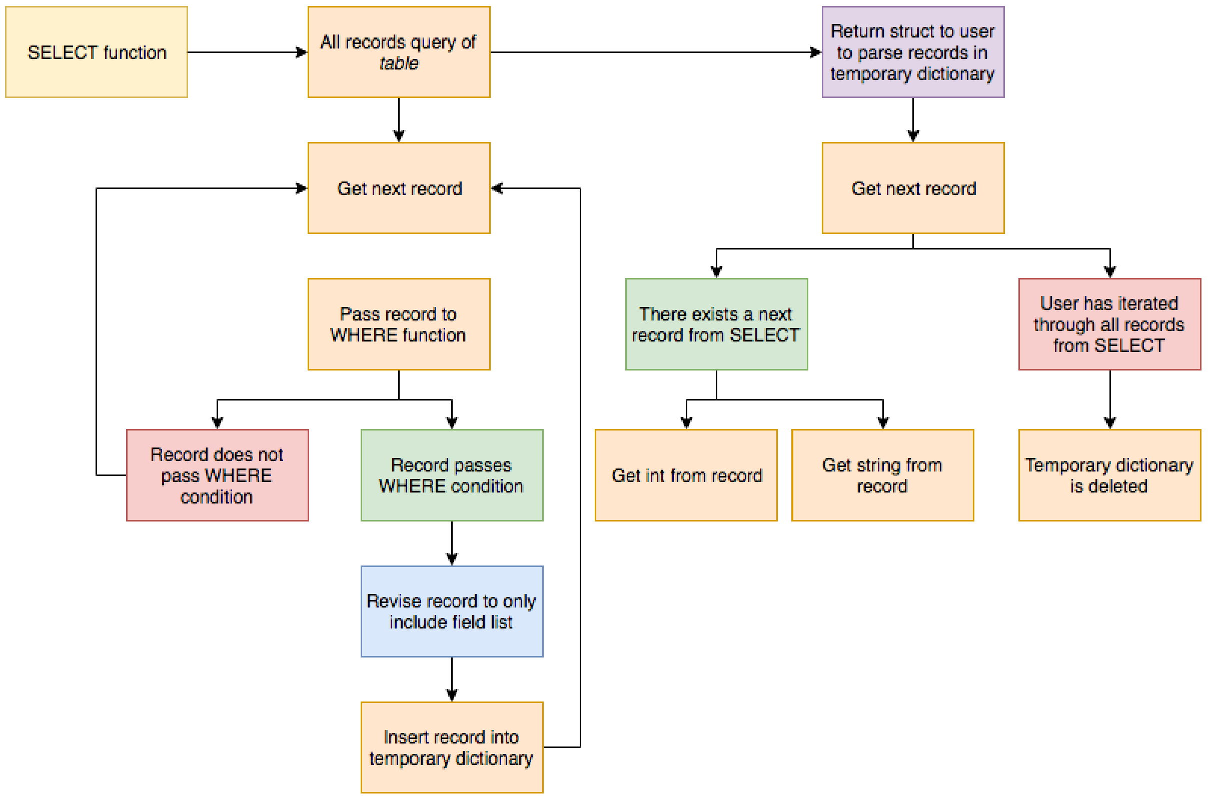

SELECT function
All records query of *table*
Return struct to user to parse records in temporary dictionary
Get next record
Get next record
Pass record to WHERE function
There exists a next record from SELECT
User has iterated through all records from SELECT
Record does not pass WHERE condition
Record passes WHERE condition
Get int from record
Get string from record
Temporary dictionary is deleted
Revise record to only include field list
Insert record into temporary dictionary

Figure 4. IINQ Select process

3.6 Drop

The simplest SQL command is arguably `DROP TABLE`, as the only user provided information needed is the table name. Following regular SQL syntax, the IINQ user code to drop a table is shown in Listing 32 below.

```
SQL_execute("DROP TABLE Dogs;");
```
Listing 32. IINQ Drop Table statement

Due to the simplicity of the command, the translated user code is one functional call with the table name as the only parameter, as demonstrated in Listing 33.

```
/* SQL_execute("DROP TABLE Dogs;"); */
drop_table("Dogs.inq");
```
Listing 33. IINQ Drop Table statement

As `drop_table` takes the table name as a parameter, the underlying code is generic and therefore can be used to destroy any table from memory.

4. Experimental Results

4.1 Testing Environment

Performance testing of IINQ commands are run on an Arduino ATmega2560 microcontroller with an Ethernet shield and microSD card. This model of embedded device is equipped with 256 KB of Flash memory and 8 KB of SRAM, not including the added capacity of the microSD card [27]. It has a clock speed of 16 MHz [27].

The following test cases have been broken into SQL commands. For each IINQ SQL function, variations of the test are also performed when applicable, such as different tests with a variable number of `WHERE` conditions. Each test follows the same table schema, with a record composed of two integers and a character array of length 30, which equals a value size of 38 bytes. Due to the user defined schema, these tests results may not be replicable with a larger record size, however, the performance may be exceeded with smaller records. Although `create_table` and `drop_table` are not explicitly included in the test cases below, they are imperative to each scenario as they facilitate the functionality of each test.

As testing is performed on a memory constrained device, we must be certain that IINQ has stringent memory management. To ensure this, testing was also performed using Valgrind [28]. This tool allowed all memory leaks to be caught and resolved, guaranteeing that IINQ is of high quality and suitable for embedded devices.

The graphs included in the testing results below all follow the same format. The x-axis shows the number of records within the created table, and the y-axis signifies the number of milliseconds elapsed while performing 5 iterations of the particular command.

4.2 Insert

4.2.1 Simple Insert Statement

This test case follows the `INSERT` statement format outlined in Listing 16. From testing we calculate the mean and standard deviation of performing 5 insertions, in the simplest case, to be 350.447 ms and 22.217 ms, respectively. Figure 35 demonstrates the testing results.

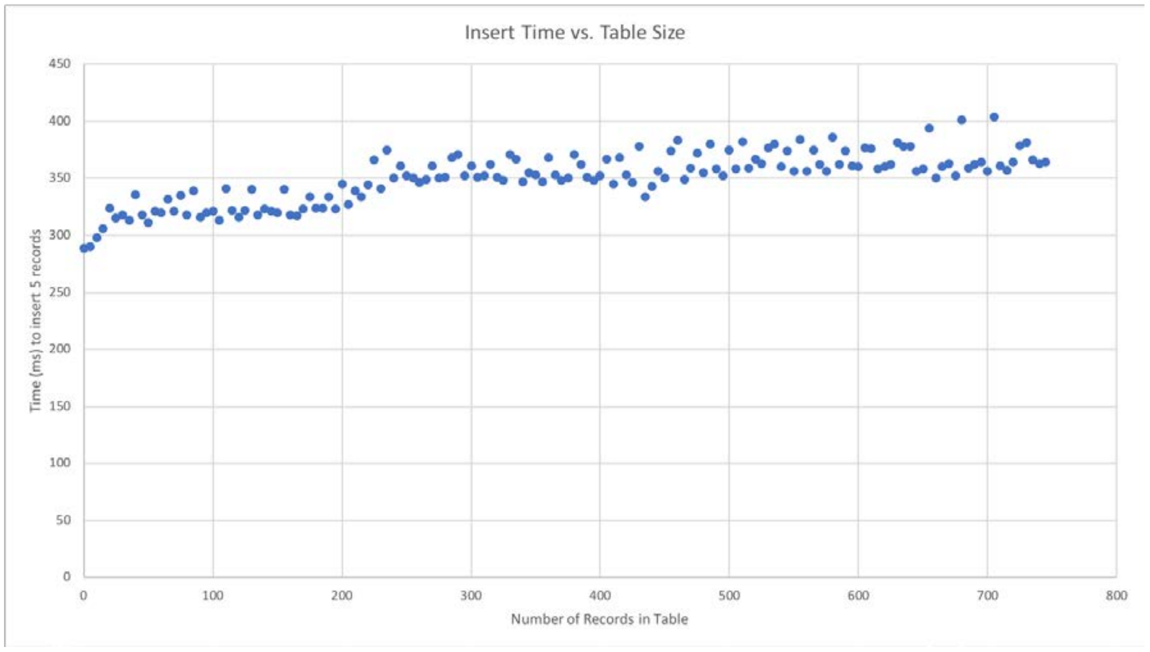


Figure 5. Performance of an insert.

It is evident from Figure 35 that as the table size increases, the time taken to perform 5 insertions does not differ drastically.

4.2.2 Prepared Insert Statement

Differing from the simple `INSERT` case demonstrated above, we also tested prepared `INSERT` statements as outlined in Listing 17. This insertion has the added step of updating the record value with a parameterized value. The mean and standard deviation of performing 5 prepared statement insertions are 354.993 ms and 47.286, respectively. Although this mean does not vary widely from the mean calculated for a simple `INSERT`, the standard deviation of a prepared statement insertion is double. This variation in the data may be due to outliers during testing. In addition, the optimization of the C code has been performed prior to run-time, therefore the two `INSERT` statements are identical with the small added step of the byte array value update for a prepared statement, which does not decrease performance time substantially. The results of this test are shown in Figure 6.

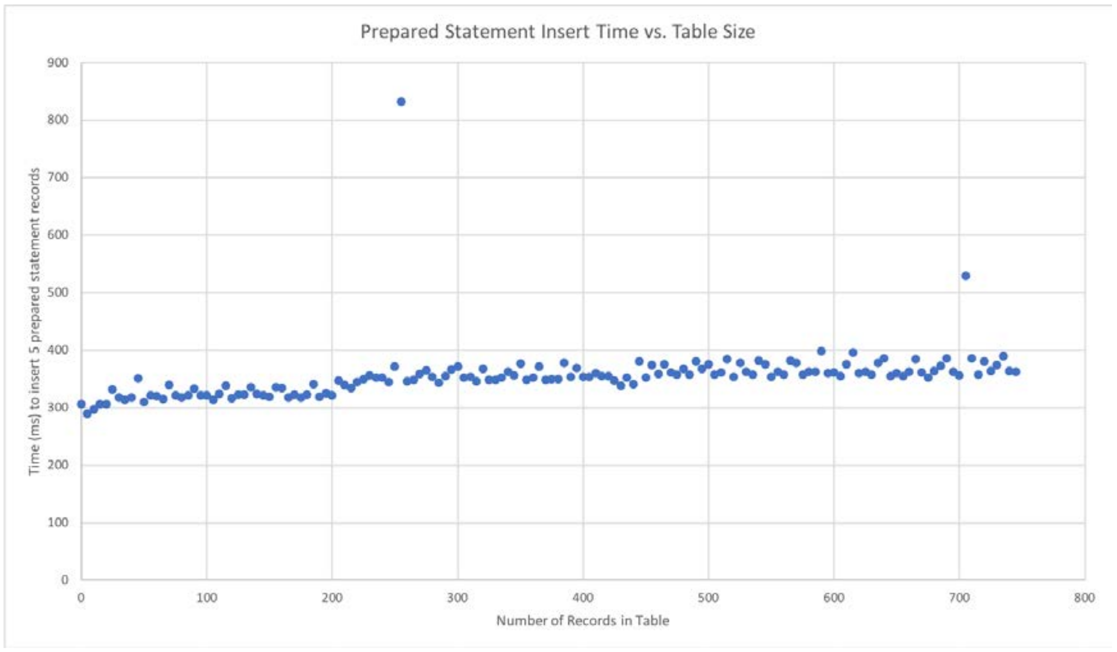


Figure 6. Performance of a prepared statement insert.

4.3 Update

Multiple tests were performed to evaluate the performance of `UPDATE`, `DELETE`, and `SELECT`, as their architecture contains additional computation and function calls. Hence, testing for these commands was repeated with variations of parameters, including the number of `WHERE` conditions and the number of fields to update.

4.3.1 Update Statement – 1 field to update, 1 where condition

This test case refers to an `UPDATE` statement with one field to update in the command, and one `WHERE` condition for the records to pass. The mean of the performance of this test is 4066.14 ms with a standard deviation of 1135.427 ms. The full results are displayed in Figure 7.

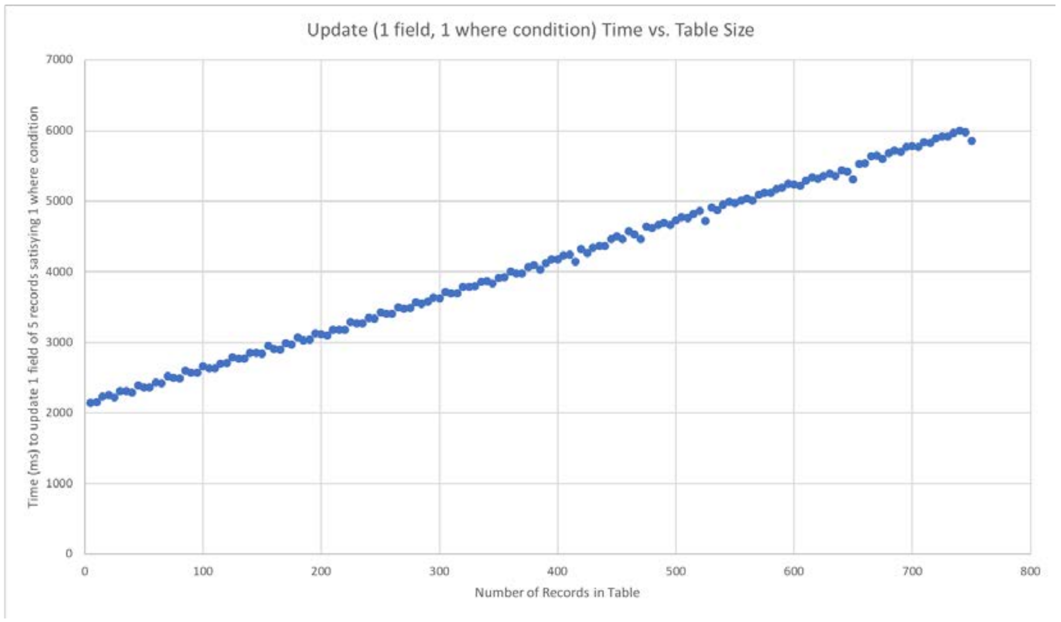


Figure 7. Performance of an update with one field, one where condition.

4.3.2 Update Statement – 1 field to update, 3 where conditions

The testing of `update` was repeated with 3 `WHERE` conditions in the command. This test executed with a mean and standard deviation of 4025.1 ms and 1118.427 ms, respectively. This mean slightly underperforms the mean calculated in Section 4.3.1, as this case is more intensive as more conditions are checked due to the additional `WHERE` conditions. The results of this test are shown in Figure 8.

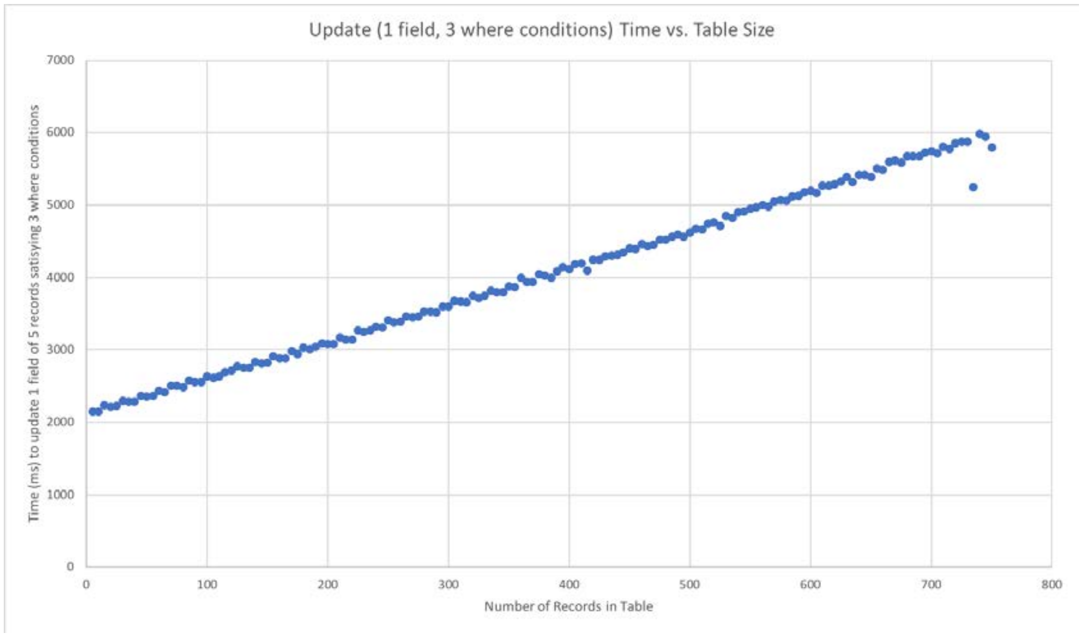


Figure 8. Performance of an update with one field, three where conditions.

4.3.3 Update Statement – 3 fields to update, 1 where condition

`UPDATE` was also tested with respect to the performance time to update multiple fields within the record. This test was repeated with varying numbers of `WHERE` conditions. The mean time to update 3 fields in a record that has met one `WHERE` condition is 4066.407 ms, with a standard deviation of 1135.149 ms. These calculations are almost identical to Section 4.3.1; therefore, it can be assumed that updating multiple fields in the record does not add much time to the performance of the command. Figure 9 shows a graph of the results of this test.

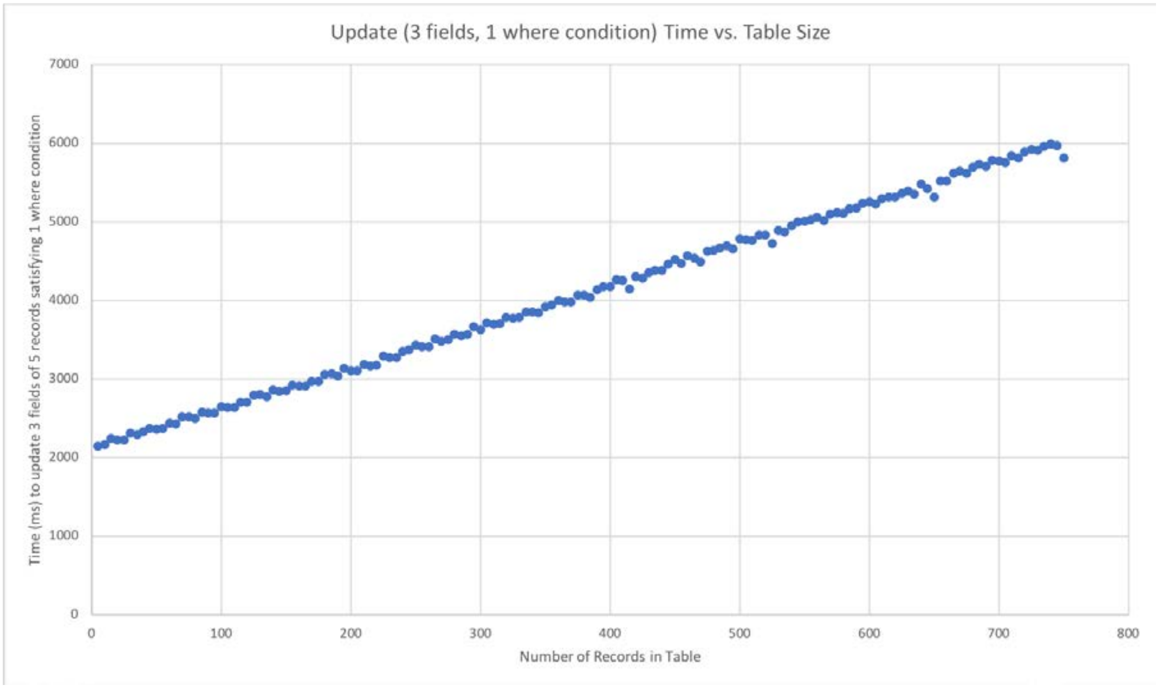


Figure 9. Performance of an update with three fields, one where condition.

4.3.4 Update Statement – 3 fields to update, 3 where conditions

To gather comprehensive testing of the `UPDATE` command, it was also evaluated with respect to updating 3 fields in a record that has passed 3 `WHERE` conditions. Although `update` could be tested further with additional fields to update and `WHERE` conditions, it is unlikely for such an occurrence to happen with regular use. However, IINQ does support updating more than 3 fields as well as more than 3 `WHERE` conditions. The mean of this test is 4199.413 ms with a standard deviation of 1197.638 ms. This mean is the largest of the `update` tests due to its computational complexity. The full test results are demonstrated in Figure 10.

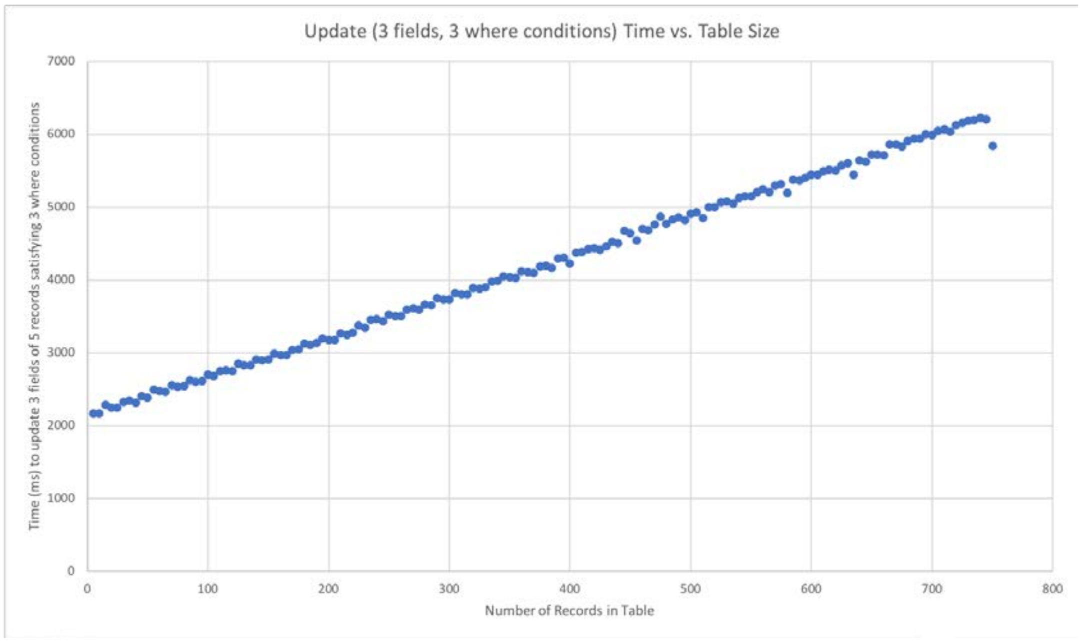


Figure 10. Performance of an update with three fields, three where conditions.

It is evident from Sections 4.3.1 – 4.3.4 that the variance of performance time of the `update` tests is much greater than that of `INSERT` testing, as exemplified by the test standard deviations. Hence, we observe that table size has a greater impact on commands such as `update`.

4.4 Delete

Testing of `DELETE` was performed in a method similar to the `UPDATE` command testing. IINQ code for performing `DELETE` commands with a varying number of `WHERE` conditions was generated and evaluated by performing the command on 5 records.

4.4.1 Delete Statement – 1 where condition

Results were gathered from performing `DELETE` on a table with varying size from 5 to 750 records. In this case, the `DELETE` statement contained one `WHERE` condition to be evaluated. This produced a mean and standard deviation of 4114.293 ms and 1181.276 ms, respectively. Figure 11 shows the full results of the test.

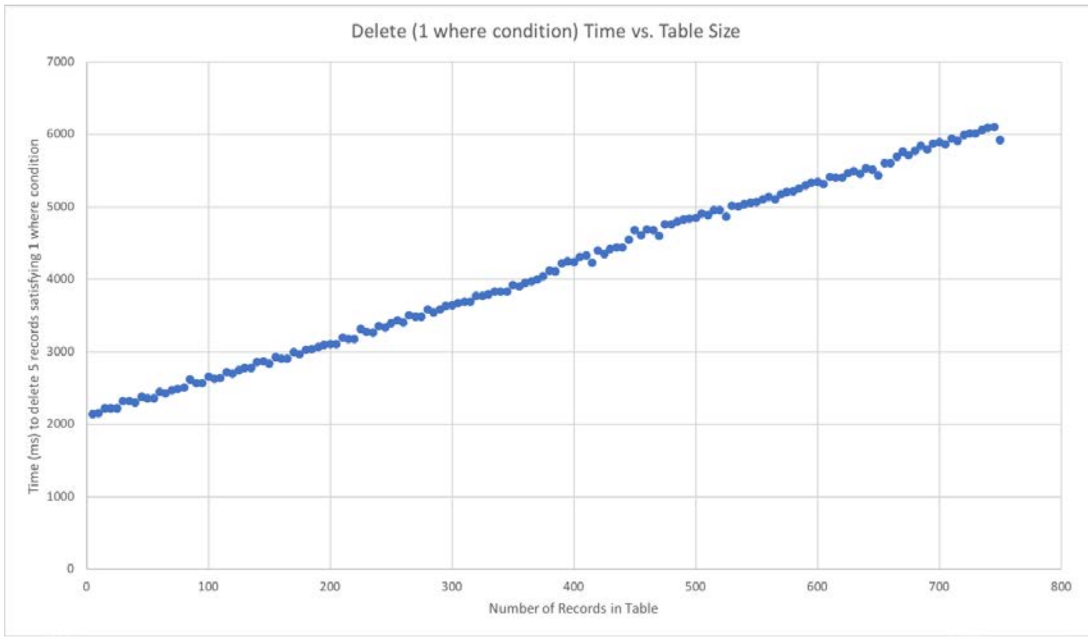


Figure 11. Performance of a delete with one where condition.

4.4.2 Delete Statement – 3 where conditions

The testing of `delete_record` was repeated with 3 `WHERE` conditions to evaluate. This generated a mean of 4128.927 ms, with a standard deviation of 1189.177 ms. More exhaustive testing for `DELETE` commands is possible through an even greater number of `WHERE` conditions, however, for regular use 3 conditions is sufficient. The results of this test are demonstrated in Figure 12.

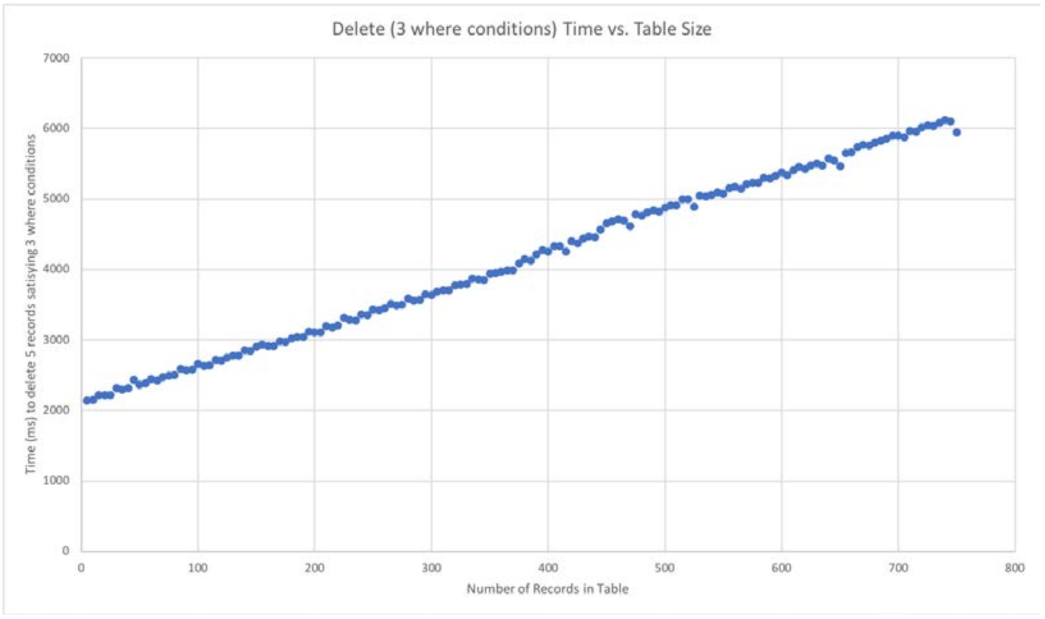


Figure 12. Performance of a delete with three where conditions.

The results of delete_record testing are comparable to update, as expected, due to their similar architecture and implementation.

4.5 Select

SELECT is the most complex SQL operator, however, IINQ only currently supports selecting a field list from a single table with WHERE conditions. Therefore, testing is performed using a varying number of WHERE conditions. The number of fields in the field list is constant at 2 as this variation causes insignificant changes in the performance of the command. In addition, getInt and getString are not included in testing as it is user responsibility how to use the returned record and does not affect the performance time of iinq_select.

4.5.1 Select Statement – select 2 fields, 1 where condition

The results gathered from testing iinq_select with a single WHERE condition and a field list of two fields produced a mean and standard deviation of 3439.193 ms and 841.821 ms, respectively. Although this mean time is greater than UPDATE and DELETE commands, it is more consistent with a smaller standard deviation. Figure 13 exhibits the full results of this test.

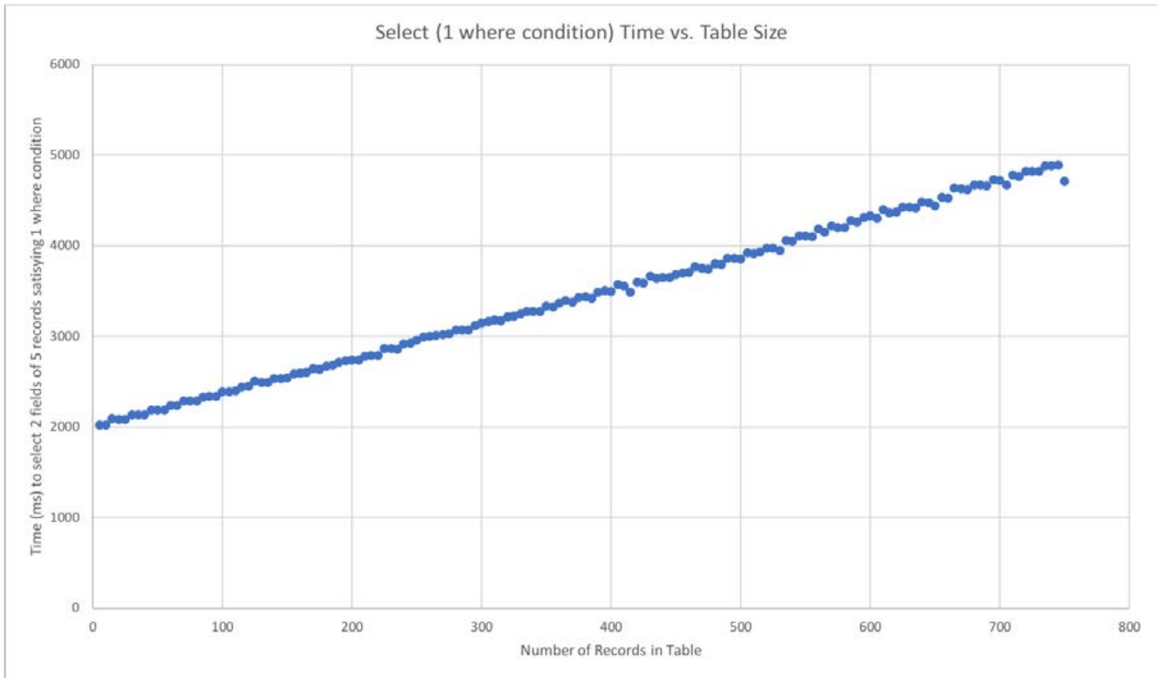


Figure 13. Performance of a select with one where condition.

4.5.2 Select Statement – select 2 fields, 3 where conditions

Comparably, testing a `SELECT` command now with 3 `WHERE` conditions produces a mean of 3470.173 ms, with a standard deviation of 856.491 ms. Hence, in comparison to Section 4.5.1, evaluating multiple `WHERE` conditions does not considerably impede the performance of `iinq_select`. The testing results are shown in full in Figure 14.

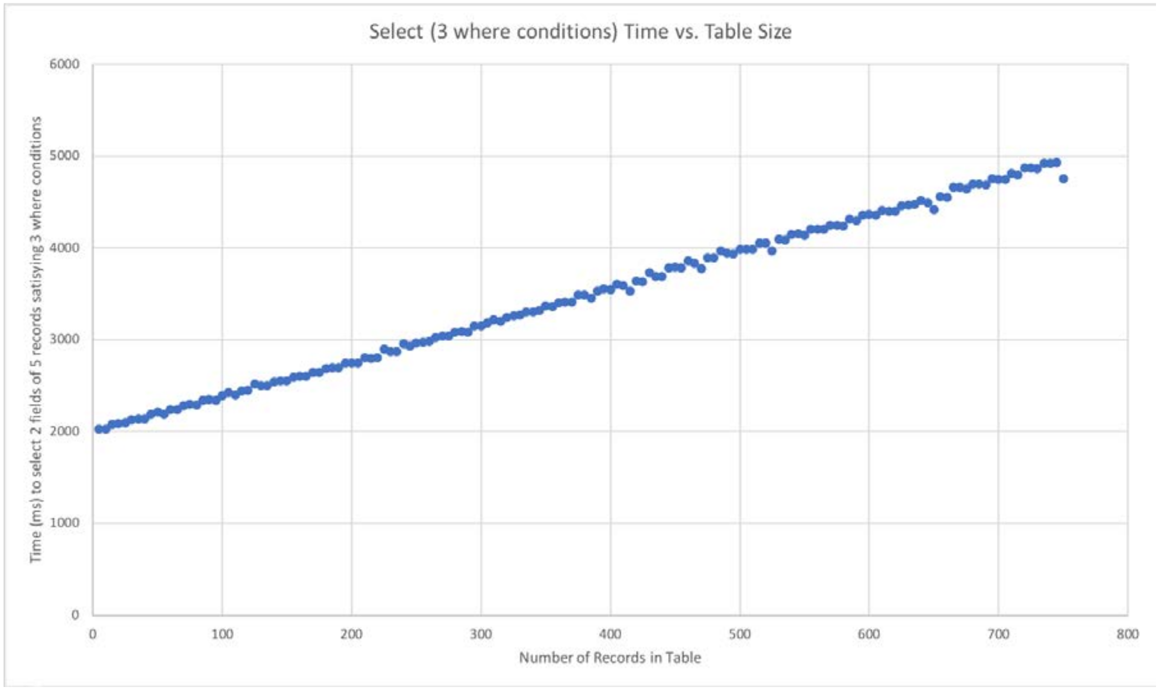


Figure 14. Performance of a select with three where conditions.

As no operations are performed on the records in the table in question during a `SELECT` command, it outperforms both `update` and `delete_record`.

5. Conclusions and Future Work

IINQ SQL syntax was created to support SQL user code while remaining performance competitive in a microprocessor environment. As these devices are memory constrained, the SQL Java translator of IINQ only includes the source code for the functions immediately called in the user code. IINQ also accommodates users by supporting SQL syntax as closely as possible, while complying to the C language structure. Development was impeded by alternating between the translation of the user code, while parsing through the Java translator, and the functionality of the generated C code. However, the concealment of all complex code from the user ensures that IINQ is as user-friendly as possible.

With further development, IINQ could be extended to include other important features of `SELECT`. These include aggregate operators, `GROUP BY` clauses, and `ORDER BY` clauses, for example. To truly replicate a relational database, these additional keywords are essential for complex queries. As demonstrated through intensive testing, this query language accomplishes the goals put forth to utilize the functionality of IonDB while adhering to the SQL standard, in a convenient and performance competitive manner.

References

[1] G. Milener, C. Rabeler and C. Guyer, "Structured Query Language (SQL)," 19 January 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/odbc/reference/structured-query-language-sql. [Accessed 27 February 2018].

[2] D. Lorentz and J. Gregoire, "Oracle Database SQL Reference," December 2003. [Online]. Available: https://docs.oracle.com/cd/B13789_01/server.101/b10759/intro001.htm. [Accessed 22 February 2018].

[3] P. Garner and J. Mariani, "Learning SQL in steps," *Journal of Systemics, Cybernetics and Informatics,* vol. 13, no. 4, p. 19, August 2015.

[4] IonDB Project, "IonDB," [Online]. Available: http://iondb.org. [Accessed 20 February 2018].

[5] D. Box and A. Hejlsberg, "LINQ: .NET Language-Integrated Query," February 2007. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb308959.aspx. [Accessed 4 March 2018].

[6] G. Harshini and M. R. Sreeha, "Internet Of Things And Analytics," *International Journal of Scientific & Technology Research,* vol. 6, no. 08, p. 287, August 2017.

[7] Office of the Privacy Commissioner of Canada. Policy and Research Group, & Canadian Government EBook Collection, The internet of things: An introduction to privacy issues with a focus on the retail and home environments, Gatineau: Policy and Research Group of the Office of the Privacy Commissioner of Canada, 2016.

[8] J. M. Hughes, Arduino: A Technical Reference, Sebastopol: O'Reilly Media, 2016.

[9] S. Fazackerley, E. Huang, G. Douglas, R. Kudlac and R. Lawrence, "Key-value store implementations for Arduino microcontrollers," in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE),* Halifax, 2015.

[10] IonDB Project, "Basic User Interface," 6 July 2017. [Online]. Available: https://github.com/iondbproject/iondb/wiki/Basic-User-Interface. [Accessed 4 March 2018].

[11] S. D. J. MacBeth, "Linear Hashing for Flash Memory on Resource-Constrained Microprocessors," University of British Columbia - Okanagan, Kelowna, 2017.

[12] W. Penson, E. Huang, D. Klamut, E. Wardle, G. Douglas, S. Fazackerley and R. Lawrence, "Continuous Integration Platform for Arduino Embedded Software," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, Windsor, 2017.

[13] A. Beaulieu, Learning SQL, Second Edition, 2nd Edition ed., O'Reilly, 2009.

[14] M. T. González-Aparicio, M. Younas, J. Tuya and R. Casado, "Testing of transactional services in NoSQL key-value databases," *Future Generation Computer Systems-the International Journal of Escience,* vol. 80, pp. 384-399, March 2018.

[15] H. Köhler, U. Leck and X. Zhou, "Possible and certain keys for SQL," *The VLDB Journal,* vol. 25, no. 4, pp. 571-596, August 2016.

[16] P. Seshadri and P. Garrett, "SQLServer For Windows CE – A Database Engine for Mobile and Embedded Platforms," in *Proceedings of 16th International Conference on Data Engineering*, San Diego, 2000.

[17] G.-J. Kim, S.-C. Baek, H.-S. Lee, L. Han-Deok and M. J. Joe, "LGeDBMS: a small DBMS for embedded system with flash memory," in *VLDB '06 Proceedings of the 32nd international conference on Very large data bases*, Seoul, 2006.

[18] M. Rosenmuller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk and G. Saake, "FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems," in *SETMDM '08 Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, Nantes, 2008.

[19] M. Rosenmuller, S. Apel, T. Leich and G. Saake, "Tailor-made data management for embedded systems: A case study on Berkeley DB," *Data & Knowledge Engineering*, vol. 68, no. 12, pp. 1493-1512, 2009.

[20] E. Booker, "Big databases get small -- IBM DB2, Oracle to be released for mobile and embedded devices," *InternetWeek*, no. 766, p. 9, 1999.

[21] M. Fotache, "Data Processing Languages for Business Intelligence. SQL vs. R," *Informatica Economica,* vol. 20, no. 1/2016, pp. 48-61, March 2016.

[22] E. Macauley, B. Kess and C. Guyer, March 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/statements/statements.

[23] K. Morton, K. Osborne, R. Sands, R. Shamsudeen and J. Still, Pro Oracle SQL, Second Edition, 2nd Edition ed., New York: Apress, 2013.

[24] D. Laudenschlager, C. Guyer, G. Milener and R. Byham, "SELECT (Transact-SQL)," 24 October 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql. [Accessed 22 February 2018].

[25] E. Macauley, R. Byham and C. Guyer, "DROP TABLE (Transact-SQL)," Microsoft, 12 May 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/statements/drop-table-transact-sql. [Accessed 3 April 2018].

[26] Oracle, "Processing SQL Statements with JDBC," [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html. [Accessed 4 April 2018].

[27] Arduino, "Arduino MEGA 2560 & Genuino MEGA 2560," [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardMega2560?setlang=en. [Accessed 6 April 2018].

[28] Valgrind Developers, "Valgrind," Valgrind, [Online]. Available: http://valgrind.org. [Accessed April 10 2018].