Efficient Querying on Networked Embedded Devices

by

Kyla Seline Reid

Supervisor: Dr. Ramon Lawrence

The Irving K. Barber School of Arts and Sciences

Undergraduate Degree in Bachelors of Science

Honours Major in Computer Sciences
And
Honours Major in Mathematics

The University of British Columbia – Okanagan Campus

April 2018

Abstract

Ion Field Network is a sensor network project that creates a standardized way to store and retrieve data on microcontrollers such as Arduino boards. The goal is to make it easy to create and deploy sensor networks for use in environmental and industrial monitoring applications. The system builds upon the IonDB library, which provides a key-value store, and allows users to query the data for visualization through a web interface. The current system is limited in robustness and in flexibility on the queries that can be performed. In this research, the query parser and processor have been significantly improved to allow for more diverse queries and robust error handling. Support was also added for queries that can be processed both on the embedded device and on the web platform. This expands the capability of the system and potential use cases by leveraging the UnityJDBC Java library for query processing. Future work involves deploying the system for environment monitoring.

Table of Contents

Abstract 2
Table of Contents 3
Table of Figures 5
Acknowledgements 6
1. Introduction 7
2. Background 7
2.1 Ion Field Network 7
2.2 Moquette MQTT Broker 8
2.3 Node-Java 8
2.4 Embedded Devices 9
2.5 Arduino 9
2.6 Unity JDBC 10
2.7 IonDB 11
2.8 Related Work 11
3. Implementation 13
3.1 Ion Field Network Components 13
3.2 User Interface 14
3.3 Query Processor 15
3.3.1 Bridge between Java and JavaScript 15
3.3.2 Create Table 16
3.3.3 Insert into a Table 17
3.3.4 Select All 18
3.3.5 Select with specifications 21
3.3.6 Delete All 25
3.4 Server 25
3.5 Broker 26
3.6 Arduino Board and Software 26
4. Walkthrough 26
4.1 Library Requirements 26
4.2 Setup Instructions 27

4.3 Server and Moquette 28
4.4 Arduino 29
4.5 Web Site 30
5. Future Work 31
6. Conclusion 31
Bibliography 33

Table of Figures

Figure 1: Ion Field Network System Architecture [2] 8
Figure 2: Arduino Board - Mega2560 [7] 10
Figure 3: Ion Field Network Use Case Diagram [2] 14
Figure 4: New robust implementation of the Query Processor 15
Figure 5: Example of creating a Java Array List with JavaScript code using Node-Java 16
Figure 6: Create Table methods to translate users query to IonDB dialect 17
Figure 7: Previous implementation of INSRT INTO using string manipulation 18
Figure 8: New implementation of INSERT INTO using UnityJDBC 18
Figure 9: Previous implementation of Select using string manipulation 19
Figure 10: The resulting Logical Query Tree from the SQL Query 'select year, temp from Temperature' 19
Figure 11: Parsing Select queries with UnityJDBC and getting the Logical Query Tree 20
Figure 12: Function to find all Table Names present in a Logical Query Tree produced from a Select Query parsed with UnityJDBC 21
Figure 13: Result Set containing two tables, returned from the Arduino Board 22
Figure 14: Part of the function to build a Schema for a table with the result returned from the Arduino 22
Figure 15: Code to build an Array List populated with tuples from the database 24
Figure 16: Arduino code to support the Delete All function 25
Figure 17: Starting up the Server with Yarn 28
Figure 18: Starting up the Moquette broker 29
Figure 19: Serial Monitor showing the Microcontroller is connected 29
Figure 20: Users web portal after some queries have been sent to the Arduino 30
Figure 21: Server Terminal after a user has created a few queries for Arduino3 30
Figure 22: Arduino's Serial Monitor after a user has created a few queries for Arduino3 31

Acknowledgements

I have been inspired and driven by all of the people who have placed their confidence in me. I would like to say thank you to my family for all their support. I am more grateful then I can express to my Dad for always trying to help whenever I need it, even if he did not fully understand what I was working on. Thank you Mom, for always thinking I'm much smarter then I really am and for teaching me to trust in my intuition. Thank you Mariah, for reading through thirty three pages of my thesis while at work. Thank you Bronson, for hanging out with me when I was super stressed from school and work. I would also like to say thank you to Megan Kurz, my classmate for always pushing me to accomplish more, possibly due to our competitive natures. Finally I would like to thank Dr. Ramon Lawrence, firstly, for taking me on as his honours student and believing in me and secondly, for all his time and help throughout the year. I would not have made it here if it were not for all these amazing people to help and push me along the way.

1. Introduction

Storing large amounts of data is becoming more common and with that comes the need to access this data and analyze certain aspects of the information. The data that is stored in a database is only useful if there is a way to call up that data and be able to view it properly, and understand what data is being displayed. Bad data, at best, costs you lots of time cleaning it up and at worst means decisions based on completely flawed and inaccurate results [1].

Ion Field Network is an application created to process and store data on microprocessors using efficient data structures for flash memory. The data will then be displayed to users on a simple web page interface. The server that will handle the requests to carry out the messages runs Moquette, an open source message broker that uses the MQTT protocol. Moquette will facilitate transfer of messages from the server to the appropriate Arduino, wait for the result then transfer it to the server.

Users need to be able to type in queries and get accurate organized results back. As well the application should be robust enough as to handle complex queries and small syntax errors. Users also should not be required to type in all their queries/statements in a certain format.

UnityJDBC provides function and SQL translation for database dialects. However, this is a Java library and Ion Field Network is an application written all in JavaScript with a small web application in HTML. The application requires a bridge API to connect with existing Java APIs before UnityJDBC can be implemented.

The contribution of this research is to build upon the Ion Field application and develop a robust query processor for a network of embedded nodes using IonDB and UnityJDBC.

2. Background

2.1 Ion Field Network

Ion Field Network is a connected network of embedded devices for the purpose of collecting, storing and processing data. The devices are currently Arduinos, which can send and receive information to a server hosted on a local PC. The local PC will perform queries on the server, translate them to a syntax that can be understood by the Arduinos, and send them back to the user specified Arduino. These Arduinos will be networked via Ethernet cables and are using a key-value data store for data management. Ion Field Network will be an interface for efficient data management for the Arduino environment.

The goal of the project is to enable networked Arduinos, managing their data with a key-value data store, to communicate with a centralized interface through an Ethernet connection. The key-value data store that is used is IonDB. It uses a relational database capable of running on an embedded device with no more than 4KB of memory.

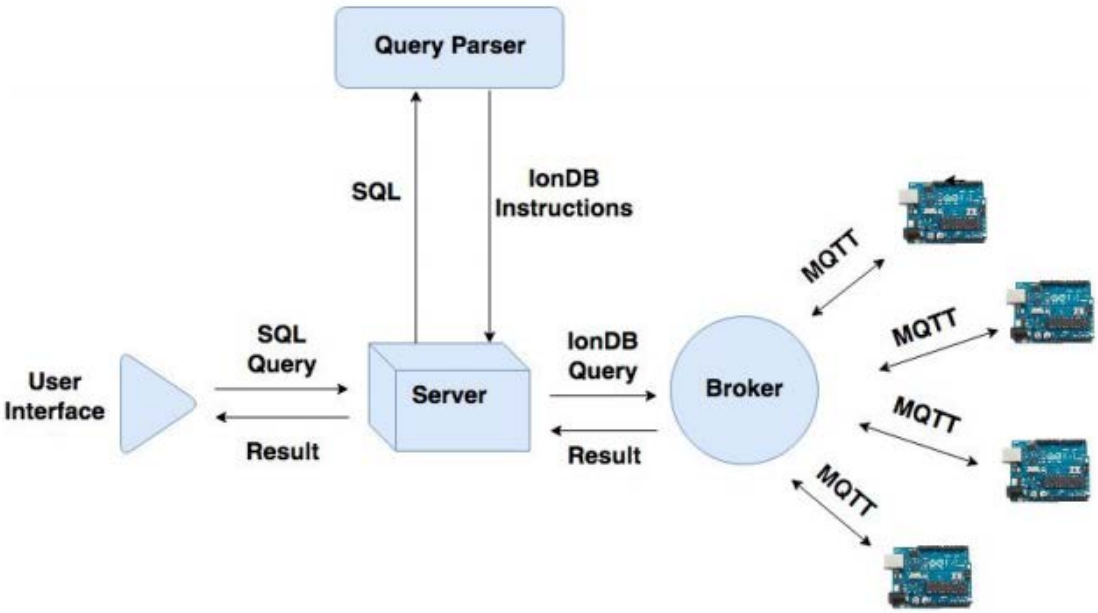

Figure 1: Ion Field Network System Architecture [2]

2.2 Moquette MQTT Broker

Moquette is an open source message broker that uses the MQTT protocol. The MQTT protocol is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium [3]. MQTT was invented by Dr Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom in 1999.

Moquette is used for machine-to-machine communication and monitoring needs of remote controlled devices for developers of the Internet-Of-Things. Moquette is a simple and small self-contained Java implementation of an MQTT broker.

The Moquette broker is lightweight and easy to understand so it can be embedded in other projects without much difficulty. By default it lives standalone, but could be integrated into an OSGi container to create more significant integrations [4].

2.3 Node-Java

Free software under the MIT copyright licence available on Github at https://github.com/joeferner/node-java, created as a bridge API to connect with existing Java APIs. With this users can create and make calls to Java objects within their existing JavaScript application. It is possible to create synchronous as well as asynchronous methods that return promises, by setting a property of the Java object.

Node-Java requires the user to install Java in the project directory then create a Java object in the JavaScript code. This Java object can then be used to create any Java object, for example, ArrayLists, and use their functionality/methods within JavaScript. There are some limitations though, for instance, users cannot create objects from abstract or interface classes. Also users also cannot use any static methods of an object.

2.4 Embedded Devices

Embedded devices have been growing in popularity over the years, with the decrease in size and cost of microprocessors. The computer completely encapsulates the device it controls in an embedded system; they are known as special-purpose systems. An embedded system is a computer system with a dedicated function usually within a larger mechanical or electrical system. It performs pre-defined tasks, usually these tasks have very specific requirements.

They can be embedded as part of a complete device, often this is done in hardware and mechanical parts. Embedded systems control many devices in common use today [5]. For example, modern-day cars use advanced embedded systems to perform many functions which were previously analog. The tasks can range from how music is played and controlled to the ignition. An increasing number of domains are utilizing embedded devices. Consumer cooking, industrial, automotive, medical, commercial, military applications and national infrastructure such as telecommunications rely heavily on the capabilities provided by embedded devices.

2.5 Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software. It's intended for anyone making interactive projects. It consists of an Arduino board and Arduino software. The Arduino board can be used for sensing the environment it is in, through inputs from its sensors, and for affecting the environment through lights, motors and other actuators. It is an inexpensive programmable circuit board. The Arduino software is used to tell the Arduino what to do, through code written in the Arduino programming language using the Arduino development environment [6]. This Arduino IDE for creating Arduino projects is available on MacOS, Windows, and Linux operating systems.

Arduinos provide a means for people with little knowledge about electronics to develop code and embedded hardware projects. The Internet of Things has seen a lot of Arduino-driven development, as Arduinos allow less experienced developers to build internet-enabled devices capable of performing networked operations. In the past, projects using embedded devices were usually only created by people with knowledge of electrical engineering. Arduinos have greatly increased the number of people that can now participate in the Internet of Things projects. As such the benefits of the Arduino project have been numerous for developers wanting to get started in the Internet of Things.

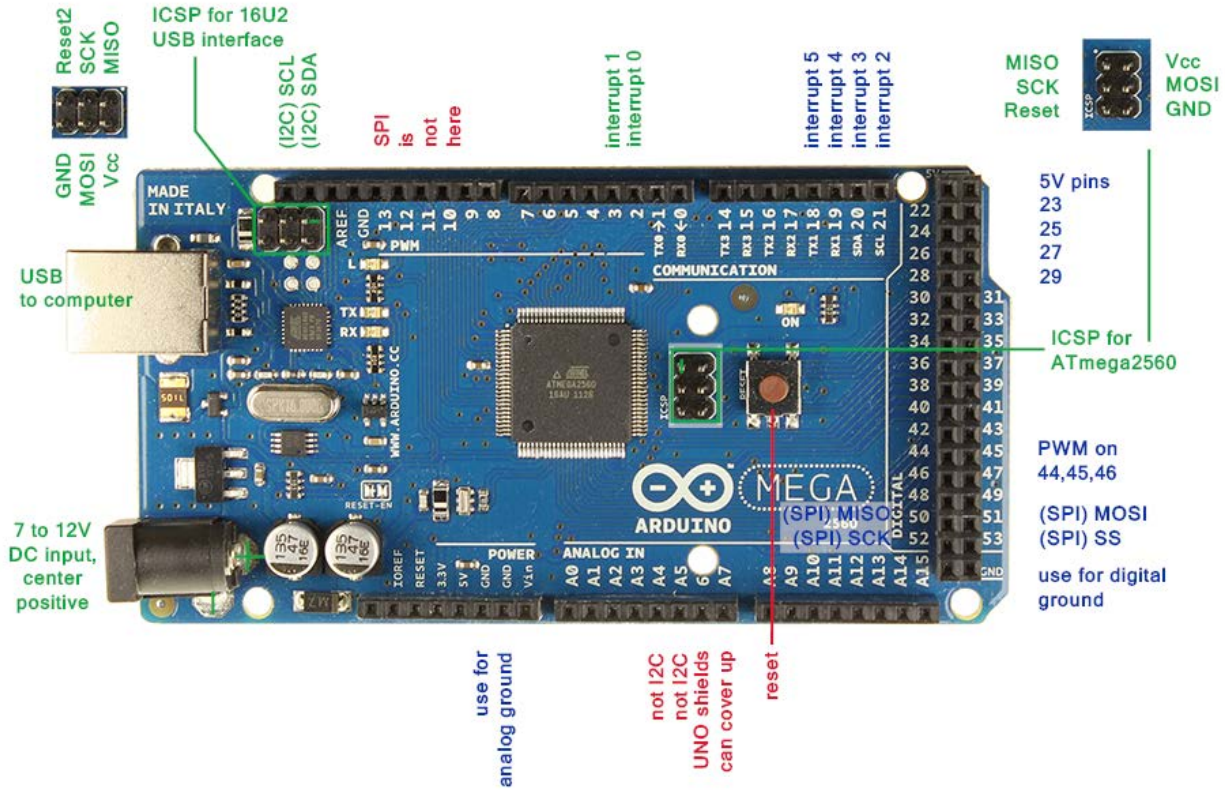

Figure 2: Arduino Board - Mega2560 [7]

2.6 Unity JDBC

UnityJDBC is a Java application that offers simple data virtualization and data processing. It was designed to make data virtualization faster, easier, and less costly. It provides function and SQL translation for database dialects. Users can type out queries in multiple ways and UnityJDBC will translate them to standardized SQL queries. Users may develop their application with one SQL dialect, and UnityJDBC will automatically translate it for whatever database is used. UnityJDBC allows any Java query and reporting software to combine data from multiple databases without requiring any new systems or server modifications [8]. UnityJDBC is a standards compliant JDBC driver that can be used in any reporting system or query software. It is available as a plug-in to SQuirreL SQL, an open source query tool. UnityJDBC was created over the course of ten years by Dr. Ramon Lawrence at the University of British Columbia Okanagan and is constantly being developed upon. The product has also been featured in multiple published peer-reviewed articles.

2.7 IonDB

IonDB is a data management system designed for Arduinos which presents a library of implementations of key-value stores for use on Arduino devices. IonDB makes it easy for Arduino programmers to manipulate data without worrying about implementing data structures and query libraries. IonDB was created for the purpose of expanding the functionality of the Arduino platform by creating an efficient interface for data organization specifically for memory constrained devices. Databases improve programmer efficiency by providing abstraction from storing and querying data. Unfortunately, not even the least computationally demanding relational database software, SQLite, can run on the Arduino platform.

There is a great performance advantage to being able to process data close to its source, this reduces latency and communication costs. This becomes even more beneficial when a number of devices are connected to each other and continually processing more data. Usually data collection devices are just simple microcontrollers with limited memory and CPU resources [9]. Previous relational databases have been created for microcontrollers such as LittleD [10], PicoDBMS [11], TinyDB [12] and Antelope [13], though the device's limitations restrict the performance capabilities and what features can be supported. IonDB uses the approach of key-values stores; this is a class of data storage system with a simple query interface but no structured query language. Elements are referred to by a key value, and they can take many different forms.

IonDB offers the first implementation of key-value stores for embedded processor platforms. It uses several data structures to implement a number of key-value stores that can be deployed on an Arduino. The library uses the same API for all the data structures and can be easily extended to new data structures and algorithm variations [9].

Currently there are two memory-based implementations and two file-based persistence implementations for Arduino microcontrollers. These include a skip-list, flat-file, and disk- and main-memory-based implementations of a hash map. All of the implementations support insert, update, get, and delete operations, as well as find and range queries. IonDB has a flexible and dynamic framework to swiftly deploy and change the key-value store's underlying storage structure as well as performance constraints. There is an interface that allows users to choose a structure that addresses the runtime and storage requirements of their application and to store and query data on the device.

2.8 Related Work

This is not the first attempt at creating a network of embedded devices with the capability of querying information. As sensor networks are becoming more affordable and widely deployed in a number of different applications and uses, there have been a number of attempts to create databases and query languages for such cases. Some similar works include TinyDB [12], Cougar [14] and GOptimaEmbed [15] which are networked database systems.

TinyDB, created by Madden, Franklin, Hellerstein and Hong is an acquisitional query processing system for sensor networks. TinyDB is written all in Python for the purpose of large distributed networks of sensor devices, which are usually embedded into some physical environment where they monitor and collect data. TinyDB is a distributed query processor that runs on each of the nodes in a sensor network, and takes the approach of acquisitional query processing, rather than just trying to be energy efficient. Smart sensors can control where, when, and how often data is physically acquired and delivered to query processing operators. TinyDB takes advantage of this new query processing opportunity. They focus on the locations and costs of acquiring data, which leads to significantly reduced power consumption. Queries in TinyDB, consist of a SELECT-FROM-WHERE-GROUPBY clause supporting selection, join, projection, and aggregation, like is SQL. The queries are submitted at a powered PC, parsed, optimized and sent into the sensor network, where they are disseminated and processed, with results flowing back up the routing tree that was formed as the queries propagated [12].

Cougar is a model for sensor databases, which introduces a new concept of a sensor database system, where the queries dictate which data is extracted from the sensors. In Cougar the stored data is represented as relations. While each type of sensor is modeled as a new Abstract Data Type and the sensor data is represented as time series. Signal-processing functions are modeled as abstract data type functions that return sensor data. Long-running queries are written in SQL and there is little modification to the language. Each long-running query formulated over a sensor database defines a persistent view, which is maintained during a given time interval [14].

GOptimaEmbed, created by Osegi, and Enyindah, presents a novel smart database management system for the purpose of intelligent querying of databases in device constrained embedded systems. It follows from the need to realize real-time data acquisition solutions that will facilitate information retrieval from small data repositories as well as real time data from production stations. GOptimaEmbed implements genetic algorithms as the main search engine as well it uses a model, based on an invented device dependent Short-messaging-Structured Query Language (SMS-SQL) schema translator to simplify the query process. With the use of a genetic algorithm optimized solution, one can possibly attain the solution state earlier than conventional systems. As using the approach of a genetic algorithm has the advantage that they can serve as very good approximate reasoners and converge to the local minima quickly and effectively. GOptimaEmbed uses device SQL which is SQL for an embedded processor database. This database is often of the Array structure and builds on the internal memory of the device in question [15].

3. Implementation

3.1 Ion Field Network Components

The Ion Field Network implements an interface for sending and receiving data from a network of Arduinos. Connected Arduinos send and receive information to a server hosted on a PC. Users of Ion Field will have access to a web interface application, where they can type out queries to view the appropriate information from selected Arduino boards in the network. These queries are first picked up by the server. In order to process the SQL queries sent to an Arduino, the queries must first be translated to a syntax IonDB can understand, as IonDB is the database management system in use. Therefore, the server runs a query processor which uses UnityJDBC with the help of Node-Java to bridge between the JavaScript application (which runs the query processor) and the Java library (UnityJDBC). After translation the queries must be sent over the network to the user specified Arduino, then the proper results must be returned to the user. To send the SQL queries to their specified Arduino's a "broker" is used; this "broker" is Moquette and it runs using the MQTT protocol. From there the proper statements and results can be sent and received from the Arduino board running IonDB.

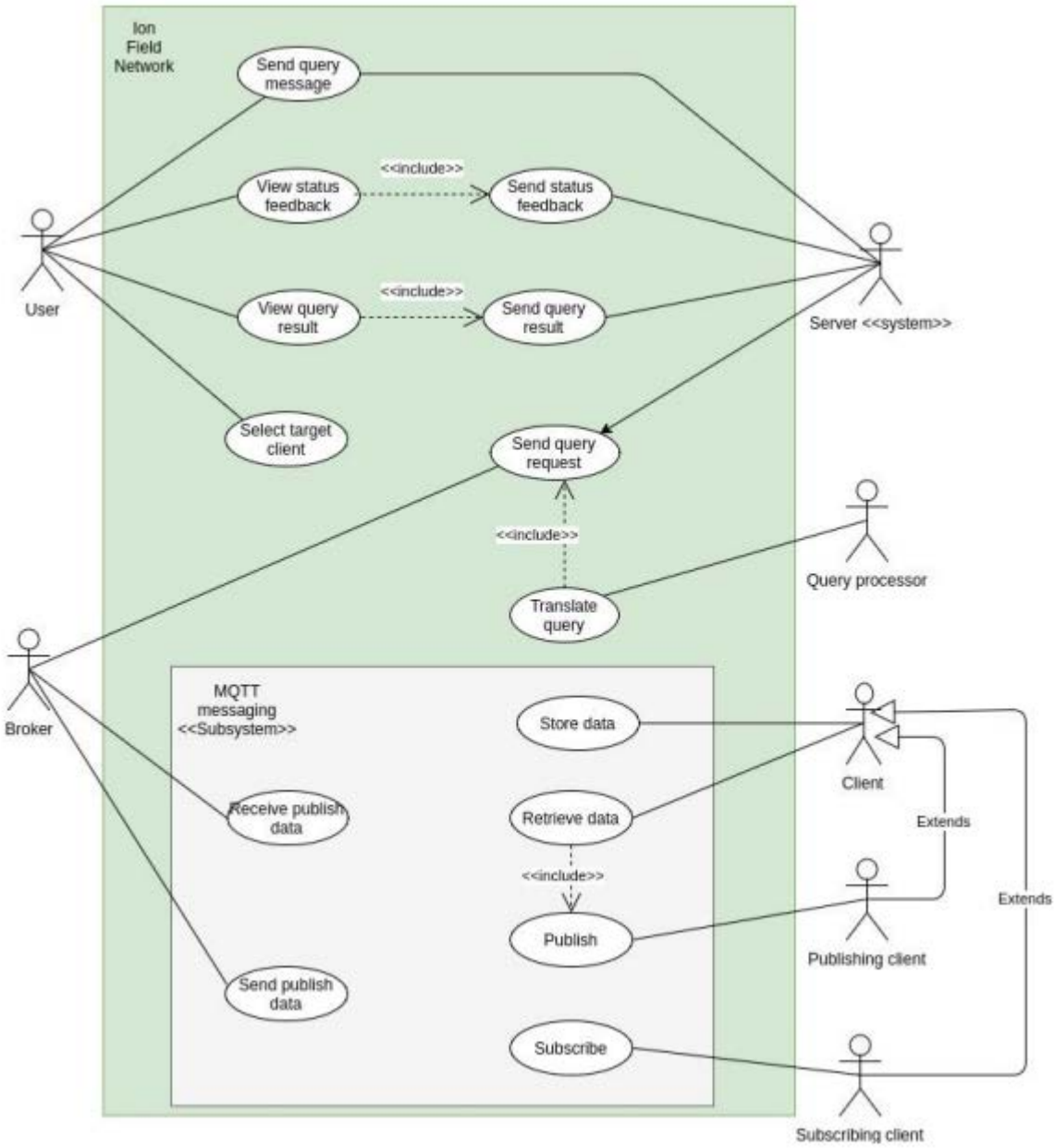

Figure 3: Ion Field Network Use Case Diagram [2]

3.2 User Interface

The user interface consists of a small web portal built using HTML and CSS, and runs on the client's personal computer. It is simple to understand and properly use. The web site contains a textbox where users may type out their queries and a result box where all the subsequent results from the queries are displayed to the user. As well there is a section at the bottom where all the user's past queries are stored and displayed so users may view their history of queries. Whenever

an Arduino board is connected to the server through the broker its name will appear on the web site along with a check box, so that users may choose which boards to send their queries to.

3.3 Query Processor

The Query parser is written with JavaScript and is used by the server to parse the user's queries and then translate them to IonDB language. This is done with the help of UnityJDBC and is built up of a few parts.

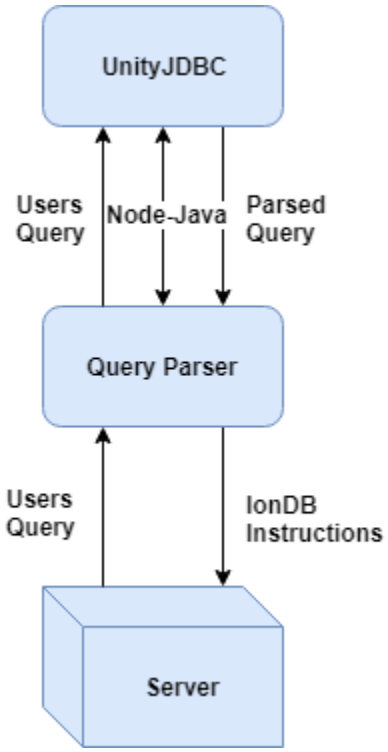

Figure 4: New robust implementation of the Query Processor

3.3.1 Bridge between Java and JavaScript

UnityJDBC is a Java library, so a bridge is implemented within the query processor so that the library may be used in the JavaScript code. This bridge is created with Node-Java, which allows instances of Java objects to be created and used within JavaScript code. The package first had to be installed in the application then a directory called BridgeJava was created in the project folder with multiple classes to implement Node-Java. Once you add a Java library into the application its objects and their functions can then be used. This made it possible to use UnityJDBC objects and their methods within the project and greatly improve the functionality of the query parser.

```
var java = require("../bridgeJava");
java.classpath.push("unityjdbc.jar");
var ArrayList = java.import('java.util.ArrayList');
var array = new ArrayList();
array.addSync("Temperature");
```

Figure 5: Example of creating a Java Array List with JavaScript code using Node-Java

3.3.2 Create Table

A user can create a new table in the IonDB database to store information in.

Example:

```
CREATE TABLE Temperature (day int, month String, year int, temp int);
```

Unfortunately, the UnityJDBC library does not support create table queries. Therefore the create table class is implemented using string manipulation to translate the users query to a language IonDB can understand. A semicolon is added to the end of the string if the user did not type one, the substring 'create table' is sent all to lower case and any unneeded spaces are removed if present in the list of attribute names and types. This is to make all the queries of similar structure no matter how a user typed it out in the web interface.

From this new updated SQL string, the table name is then pulled out and assigned to a constant variable. The list of attribute names and each of their data types is then inserted into an ArrayList. Three methods have been made to convert this ArrayList into IonDB dialect with the key as the attribute name and the value as the attribute data type.

Example

```
{"op_code":"c","query":{"table":"Temperature","fields":"day:i;month:s;year:i;temp:i;"}}
```

This is the string that would be sent to the Arduino.

```
function compressFields(fieldString) {
  let fieldData = '';
  const fields = fieldString.split(',');
  fields.forEach(field => {
    const [ fieldName, fieldType ] = field.split(' ');
    const fieldDatum = compressFieldElements(fieldName, fieldType);
    fieldData = appendFieldDatum(fieldDatum, fieldData)
  });
  return fieldData;
}

function appendFieldDatum(fieldDatum, fieldData) {
  return `${fieldData}${fieldDatum};`; // add field data to data string
}

function compressFieldElements(fieldName, fieldType) {
    const typeCode = fieldType[0].toLowerCase();
    return `${fieldName}:${typeCode}`;
}
```

Figure 6: Create Table methods to translate users query to IonDB dialect

3.3.3 Insert into a Table

Users can insert data/information into a specified table in the IonDB database. Currently there is no way to make any attributes in a table required (or Not Null), so users can choose to add values to all the attributes or only a specified few attributes in the table.

Example

```
INSERT INTO Temperature (day, month, year, temp) VALUES (03, 'March', 2018, 15);
INSERT INTO Temperature (year, temp) VALUES (2018, 15);
```

UnityJDBC does offer support for insert queries, so string manipulation no longer needs to be used on the user's query in order to translate it to IonDB language. This makes the interface much more robust and efficient when it comes to what the users can type into the web application in order to insert information into a table on an Arduino board. Previously an error would occur when a user did not capitalize the proper words or add a semicolon to the end or did not have the proper spacing in the query string. These no longer cause an issue when UnityJDBC is used to first parse the query, making all users queries into the same structure and form no matter how they typed the query in the web application. From this newly parsed query the required information is pulled out and converted into the appropriate IonDB format.

```
module.exports = sql => {
  const tableName = sql.split('INSERT INTO ')[1].split('(')[0];
  const fieldNamesString = sql.split('(')[1].split(')')[0].split(', ').join(',');
  const fieldValuesString = sql.split('(')[2].split(')')[0].split(', ').join(',');

  function constructFieldMap(fieldNamesString, fieldValuesString) {
    let fieldData = "";
    const fieldNames = fieldNamesString.split(',');
    const fieldValues = fieldValuesString.split(',');
    for(i = 0; i < fieldNames.length; i++) {
      fieldData = fieldData + fieldValues[i] + ':' + fieldNames[i] + ';'
    }
    return fieldData;
  }
```

Figure 7: Previous implementation of INSERT INTO using string manipulation

```
var gp = new GlobalParser(false, false);
var gq = new GlobalQuery();
var node = new gu();
node = gp.parseUpdateSync(sql, null);
var plan = node.getPlanSync();
var tree = plan.getLogicalQueryTreeSync();

const fields = tree.getRootSync().getInsertFieldsSync();
const values = tree.getRootSync().getInsertValuesSync();
const tableName = tree.getRootSync().getSourceTableSync().getNameSync().split('.')[1];


function constructFieldMap(Fields, Values) {
    let fieldData = "";
    for(i = 0; i < Fields.sizeSync(); i++) {
        fieldData = fieldData + Values.getSync(i).getContentSync().toString() + ':'
            + Fields.getSync(i).toString() + ';'
    }
    return fieldData;
}
```

Figure 8: New implementation of INSERT INTO using UnityJDBC

3.3.4 Select All

Select is a very important feature for all databases, as this is how the user will visualize the data stored in the database and in this case on the Arduino board. This data needs to be displayed properly so users can understand what they are looking at in order to draw the appropriate conclusions from the information being requested. Previously the Ion Field Network only had the capability to support SELECT ALL queries, so users would receive all the information stored in a single specified table on the Arduino. As well, no filters could be applied to this data to help sort through all the information. Issues with needing the appropriate words

capitalized, the spacing and a semicolon on the end also produced errors, as string manipulations were being used on the query.
Example

```
SELECT ALL FROM Temperature;
SELECT year, temp FROM Temperature;
```

Would produce the same result sets, which was all the data stored in the table Temperature.

```
module.exports = sql => {
  const tableName = sql.split(' FROM ')[1].split(' WHERE')[0].split(';')[0];
  const fieldString = sql.split('SELECT ')[1].split(' FROM')[0]
      .split(', ').join(',');

  function constructFieldList(fieldString) {
    return fieldString.split(',')
  }

  return { table: tableName };
}
```

Figure 9: Previous implementation of Select using string manipulation

Using UnityJDBC to parse the select query from the user first, greatly decreases any syntax errors from occurring on the user's side due to how they typed the query on the web application. This is because UnityJDBC has the functionality to parse many different forms of select queries and produce the appropriate logical query trees for each.

```
PROJECT: Temperature.year, Temperature.temp
    PROJECT: Temperature.year, Temperature.temp
        Temperature (source: "DB") (size: 0)
```

Figure 10: The resulting Logical Query Tree from the SQL Query 'select year, temp from Temperature'

```
module.exports = sql => {
    var java = require("../bridgeJava");
    java.classpath.push("../unityjdbc.jar");

    var i = sql.length;
    if((sql.charAt(i-1))!== ";"){
        sql = sql+';';
    }

    var GlobalParser = java.import("unity.parser.GlobalParser");
    var GlobalQuery = java.import("unity.query.GlobalQuery");
    var GQTableRef = java.import("unity.query.GQTableRef");
    var GQFieldRef = java.import("unity.query.GQFieldRef");
    var LQTreeConstants = java.import("unity.query.LQTreeConstants");

    var gp = new GlobalParser(false, false);
    var gq = new GlobalQuery();
    gq = gp.parseSync(sql, null);
    var lqtree = gq.getLogicalQueryTreeSync();
    var root = lqtree.getRootSync();
```

Figure 11: Parsing Select queries with UnityJDBC and getting the Logical Query Tree

Using the resulting logical query tree produced from UnityJDBC, it is then possible to create a function to find all the table names present in the select query. The table names can now all be identified and stored into an array. The clause in which the table names appear in the query no longer matter either, using the query tree it is possible to properly retrieve all of them.

```
function findTables(root, tableList){
    if (root === null) return;
    var tref;
    if (root.getTypeSync() === LQTreeConstants.IDENTIFIER){
        var obj = root.getContentSync();

        if (obj !== null && java.instanceOf(obj, "unity.query.GQFieldRef")){
            tref = ((GQFieldRef)(obj)).getTableSync();
            if (!tableList.includes(tref))
                tableList.push(tref.getNameSync());
        }
    }
    else if (root.getTypeSync() === LQTreeConstants.TABLE){
        tref =  ((root.getContentSync()));

        if (!tableList.includes(tref)){
            tableList.push(tref.getLocalNameSync());}
    }
    for (var i = 0; i < root.getNumChildrenSync(); i++){
        findTables(root.getChildSync(i), tableList);}

}

var tableName=[];
findTables(root, tableName);
```

Figure 12: Function to find all Table Names present in a Logical Query Tree produced from a Select Query parsed with UnityJDBC

This list of table names is then sent to the specified Arduino board. The full result set for each table is then returned, with all the data/information that is stored in each table and displayed to the user on the web application.

3.3.5 Select with specifications

The functionality to view only the appropriate and relevant information is especially important. Users do not want to have to always view everything stored in a table and then sift through it themselves for the data they need. Changing the Select All function to filter information and only show the applicable data proved very difficult. Firstly, all the data stored in IonDB is saved as a long string of key value pairs, due to memory constraints. Therefore, when Select is called, a JSON string object is returned containing all the table information stored in key-value form. So, it is not possible to do any filtering of data on the Arduino board itself. Secondly, there are no schemas for any of the tables that are created. Thirdly, the code needed to be updated on both the Arduino side and the JavaScript side.

To tackle this problem all Select queries are first treated as Select All queries. The Array of table names is sent to the Arduino board. All the information for each table is pulled out and then they are all concatenated into one long JSON string, which is then sent to the server. The Arduino code was modified to take in an Array of table names and pull out the information for

all of them, and then concatenate each tables result string together. From this side the information can now be filtered before it is sent to the web portal for the user to view.
Example of a result with two tables returned from IonDB:

1) Table one with fields (attributes) colour as a string and num as an integer
2) Table two with fields (attributes) random as a string and id as an integer

```
"one\ncolour:s;num:i;\n'red':colour;10:num;\n'yellow':colour;8:num;\n'green':colour\n\
n;EORtwo\nrandom:s;id:i;\n'red':random;10:id;\n'water':random;8:id;\n'green':random;15
:id\n\n;EOR\u0000";
```

Figure 13: Result Set containing two tables, returned from the Arduino Board

The server processes all results returned from the Arduino board. If the query was a Select query it is then sent off to another class to be properly parsed. This new result is then returned to be published to the user. To parse the result properly so that it only displays the wanted information to a user, the result string is first split into strings of each table and that table's data. Then for each table a schema is created with UnityJDBC for each table in the result.

```
for(var i=0; i<columns.length; i++){
    SorI = getTheType(columns[i], columnsAndType);
    if(SorI == 'i'){
        dataType = 4;//Attribute.TYPE_INT
        dataTypeName = "integer";//Attribut.getTypeName(sf.getDataType())
    }
    else{
        dataType = 12;//Attribute.TYPE_STRING
        dataTypeName = "varchar";//Attribut.getTypeName(sf.getDataType())
    }
    sf = new AnnotatedSourceField();
    sf.setColumnNameSync(columns[i].split('.')[0]);
    sf.setDataTypeSync(dataType);
    sf.setParentTableSync(ast);
    sf.setOrdinalPositionSync(1);
    sf.setDataTypeNameSync(dataTypeName);
    ast.addFieldSync(sf);
}
sourceDatabase.addTableSync(ast);
gSchema.addTableIdentifiersSync(ast);
return gSchema;
```

Figure 14: Part of the function to build a Schema for a table with the result returned from the Arduino

After the schema has been created for the table, an ArrayList has to be built and populated with the table's tuples from the result string. This is what UnityJDBC will scan through to produce the proper results. To create the ArrayList a few steps are involved as it is being built from just one string containing the table information in key-value pairs. The attribute names are pulled out and stored in an Array in order along with either an 's' if it's a string data type or an 'i' if it's an integer data type. Then from the user's original parsed select query, the attributes that are needed and their order is stored in another Array. This is due to the fact that the ArrayList has to be built up with tuples where each attribute is in the same order they appear in the execution tree produced by UnityJDBC and only those attributes need to be in the ArrayList.

This may not necessarily be the same order that the attributes are stored in the database though. From here the result string that was returned from the Arduino is split and each tuple that was stored in the table is then stored in an Array. An example of this Array is ['red':colour;10:num, 'yellow':colour;8:num] where colour and num are the attribute names and 'red', 'yellow', 10, 8 is the data.

Once all this information has been gathered the ArrayList can be built up and populated with the tuples stored in the table. To build up the ArrayList, using the order of attributes from the execution tree each tuple from the database is checked to see if it contains the attribute name needed, if so the data for it is inserted in the ArrayList. If the attribute name is not present due to the fact that if no data was inserted for an attribute in a certain tuple the attribute name will not appear in the tuple at all, no NULL's are present in the database. Then a NULL is inserted in the ArrayList for that attribute. As well since the result returned from the Arduino is just one long string, using the Array of attribute names and their data type, each attribute is first checked to know whether it's supposed to be either a string or an integer data type and then cast appropriately when its inserted into the ArrayList.

```
function getRowArray(Selectresult, colOrder) {
    var ArrayList = java.import('java.util.ArrayList');
    var columnsAndType = [];
    var columns = [];
    const nameTable = Selectresult.split("\n")[0];
    const column = Selectresult.split("\n")[1].split(";");
    for (var i = 0; i < column.length; i++) {
        if( column[i] != '')
            columnsAndType[i] = column[i];
    }
    if(colOrder.includes('*')) {
        colOrder.shift();
        for (var i = 0; i < columns.length; i++) {
            colOrder.push(columns[i]);
        }
    }
    var rows = new ArrayList();
    var table = Selectresult.split("\n");
    for (var i = 2; i < table.length; i++) {
        var row = [];
        var eachColumn = table[i].split(/[:]+/);
        for (var j = 0; j < colOrder.length; j++) {
            var SorI = getTheType(colOrder[j], columnsAndType);
            if (eachColumn.includes(colOrder[j].split('.')[0]) && colOrder[j] != '') {
                var ind = eachColumn.indexOf(colOrder[j].split('.')[0]);
                if(SorI == 'i')
                    row.push(parseInt(eachColumn[ind - 1]));
                else
                    row.push((eachColumn[ind - 1]).split("'")[1]);
            }
            else row.push(null)
        }
        rows.addSync(row);
    }
    return rows;
```

Figure 15: Code to build an Array List populated with tuples from the database

After the ArrayList has been created and populated with tuples of the proper attributes needed, the ResultSetScan in the tree from UnityJDBC is then modified to an ArrayList scan. The Operators for the local query node are also adjusted accordingly using the new ArrayList scan. When this process has been finished for all the tables in the select query the query can finally be executed and the proper results established. This is then sent back to the server and displayed to the user on the web site. With this new function users can now perform many different Select queries, with joins, group bys, ordering etc. where before they could only select everything from a table. This has greatly improved the capabilities of the application.

3.3.6 Delete All

The functionality to support a query that can delete all the data from a table was developed from scratch. Previously there was no support for a delete statement of any form. This is an important additional feature as users may need to delete information from a table if it is no longer needed to save room as the database is on a memory constrained device. A new class was created with the ability to parse a delete statement with UnityJDBC. The table name is pulled out and sent to the Arduino software. Within the Arduino software a new check was added to see if the operation code is for a delete statement. A new method called deleteAll was created; a cursor is established for the table then the table name and its attribute names along with their data type are stored into a string and displayed to the client. Everything else in the table is deleted, so all the information stored in the table is deleted, but the table itself and its attributes still exist in the database.

```
char* deleteAll(char* tableName) {
  Dictionary < int, ion_value_t > *table = ((Dictionary < int, ion_value_t >*) tables->get(stringToInt(tableName)));
  if (err_item_not_found == tables->last_status.error)
    return "-1";
   Serial.println("Table gotten");
  Cursor< int, void* > *my_cursor = table->allRecords();
  char* result = (char*) malloc(1);
  result[0] = '\0';
  char* value;
  if(my_cursor->next()){
    value = my_cursor->getValue();
    result = realloc(result, (sizeof(char) * (strlen(value) + strlen(result)) + 1));
    strcat(result, value);
    strcat(result, "\n"); }

    if(my_cursor->next()){
    value = my_cursor->getValue();
    result = realloc(result, (sizeof(char) * (strlen(value) + strlen(result)) + 1));
    strcat(result, value);
    strcat(result, "\n"); }

    while (my_cursor->next()) {
    int key   = my_cursor->getKey();
    table->deleteRecord(key);
  }
  value = ";EOR";
  result = realloc(result, (sizeof(char) * (strlen(value) + strlen(result)) + 1));
  strcat(result, "\n");
  strcat(result, value);
  sendMessageToTopic(result);
  delete my_cursor;
  return result;
```

Figure 16: Arduino code to support the Delete All function

3.4 Server

The server is written in JavaScript and is used in combination with the broker to facilitate communicate between the Arduino boards and the client web portal. The server must be started up with Yarn through a terminal before the application can work. Once the server is running the user should get a message on Arduino's Serial Monitor saying it has connected. The server sits

on port 3000 and listens for any outgoing queries from the users end or any incoming results from an Arduino board through the broker. Once the server gets a query from a user it first sends the query off to be properly parsed and translated to IonDB dialect and then this message is sent to the broker where it can be directed to the right Arduino board. When the server gets a message from the broker it will get the appropriate information from the result string - which may need to be adjusted and filtered and sorted in some cases. The server then publishes the proper results to the client on the web portal.

3.5 Broker

The broker is used in conjunction with the server to send and receive messages from user to Arduinos. It directs the traffic from the server to the proper Arduino(s), and from an Arduino board to the server. It was created with Moquette which is a message broker that uses the MQTT protocol and listens on port 1883 for incoming and outgoing traffic. Moquette must be started from its directory through a terminal and be running for the application to connect and work. Once Moquette receives a query from the server it passes the message along to the specified Arduino board and waits for a response. When it receives the response containing the result from the Arduino it passes this message along to the server where it can then be displayed to the user.

3.6 Arduino Board and Software

The Arduino board in use is a Mega 2560 with an Ethernet shield attached. The board plugs into a computer for power to run and the Shield is connected to a router through an Ethernet cable to connect to the internet. The Arduino software is all written in C/C++ and uses the IonDB library to implement the database on the Arduino board. When a query is received in the IonDB dialect it checks the operation code and sends it to the proper function. Each table is stored as a string consisting of key value pairs on the Arduino in a micro SD card inserted in the Ethernet Shield. Each table has a name which is used to create a hash to a unique integer to identify that table in the database. Each row in the table is numbered starting with the tables name as 1. A cursor can be created for a specified table and used traverse the rows to retrieve the information. To insert into a table, the appropriate table's string is pulled up and the new tuple's string is concatenated onto the end of the table's string and re saved on the device.

4. Walkthrough

4.1 Library Requirements

Arduino:

- IonDB v1.2.0 (https://github.com/iondbproject/iondb)

- ArduinoJson v5.11.1 (https://github.com/bblanchon/ArduinoJson)
- Arduino MQTT v1.0.0 (https://www.eclipse.org/downloads/download.php?file=/paho/arduino_1.0.0.zip)
- ArduinoUnit (for tests) v2.2.0 (https://github.com/mmurdoch/arduinounit)

MQTT Broker:

- Moquette MQTT Broker v0.10 (https://projects.eclipse.org/projects/iot.moquette)

NodeJS:

- NodeJS v6.1 or higher (https://nodejs.org/en/download/)
- Yarn v0.27.5 or higher (https://yarnpkg.com/lang/en/docs/install/)
- Express v4.15 or higher (https://expressjs.com/en/starter/installing.html)
- Blubird v3.5 or higher (http://bluebirdjs.com/docs/install.html)
- Lodash v4.17 or higher (https://lodash.com/)
- mqtt v2.9 or higher (https://www.npmjs.com/package/mqtt)
- Moch (for tests) v3.5 or higher (https://mochajs.org/)
- Chai (for tests) v4.1 or higher (http://chaijs.com/)

Node-Java:

- Java2 v0.2.0 or higher (https://github.com/joeferner/node-java or https://www.npmjs.com/package/java2)

4.2 Setup Instructions

1. Install Moquette (URL is in the Library Requirements section)

2. Once Moquette is installed, replace the moquette.conf file in

   /path/to/moquette/location/distribution-0.10-bundle-tar/config/
   With the moquette.conf provided in the config folder of the repository. For a unix based operating system use
   mv /path/to/repo/conf/moquette.conf /path/to/moquette/location/distribution-0.10-bundle-tar/config/

3. Start Moquette by navigating to the bin folder included in the installation location you chose, and run it.
   cd /path/to/moquette/location/distribution-0.10-bundle-tar/bin
   ./moquette.sh Or ./moquette.bat

4. Install Arduino MQTT (URL is in the Library Requirements section). Then unzip the file, and place it in your Arduino library directory
   cd path/to/arduino_mqtt_library
   unzip arduino_1.0.0.zip
   mv MQTTClient path/to/Arduino/libraries/

5. Install Node-Java (URL is in the Library Requirements section), from the projects root directory using Node.js

6. Navigate to the project's root directory and run Yarn to install the necessary dependencies for NodeJS, as well as start it
   cd ~/path/to/repo/location/src/query_parser
   yarn install
   yarn start

7. Navigate to the web_portal directory and open main.html with Chrome or Firefox

8. Open the Arduino IDE, and open the client.ino file located in /path/to/repo/src/arduino_client/client/client.ino
   Ensure all dependencies are installed, and the IP address/MAC address are configured to your system's specifications

9. Build the client.ino file

10. You should now see Arduino1 in the "Available Arduinos" section on the web portal.

4.3 Server and Moquette

Open two terminals up on the computer, from one navigate to the path of the server and from the other go to the location of Moquette. Start up the server with Yarn start and start up the broker with ./Moquette.bat if you are using Windows and ./Moquette.sh if you are using a Mac.

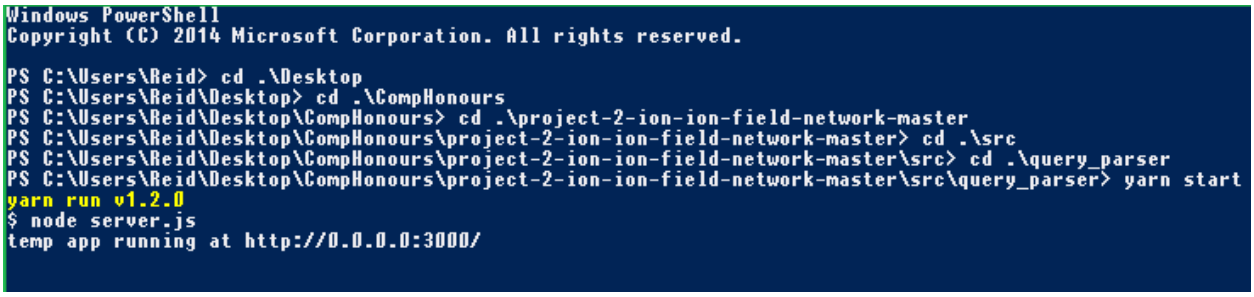

Figure 17: Starting up the Server with Yarn

Figure 18: Starting up the Moquette broker

4.4 Arduino

Make sure the Arduino board is plugged into the computer and connected to a router through an Ethernet shield. Open the Arduino client.ino class and change the IP Address to your computers IPv4 address. Then open the Serial Monitor and upload the class onto the device, a message saying connected should pop up on the Serial Monitor.

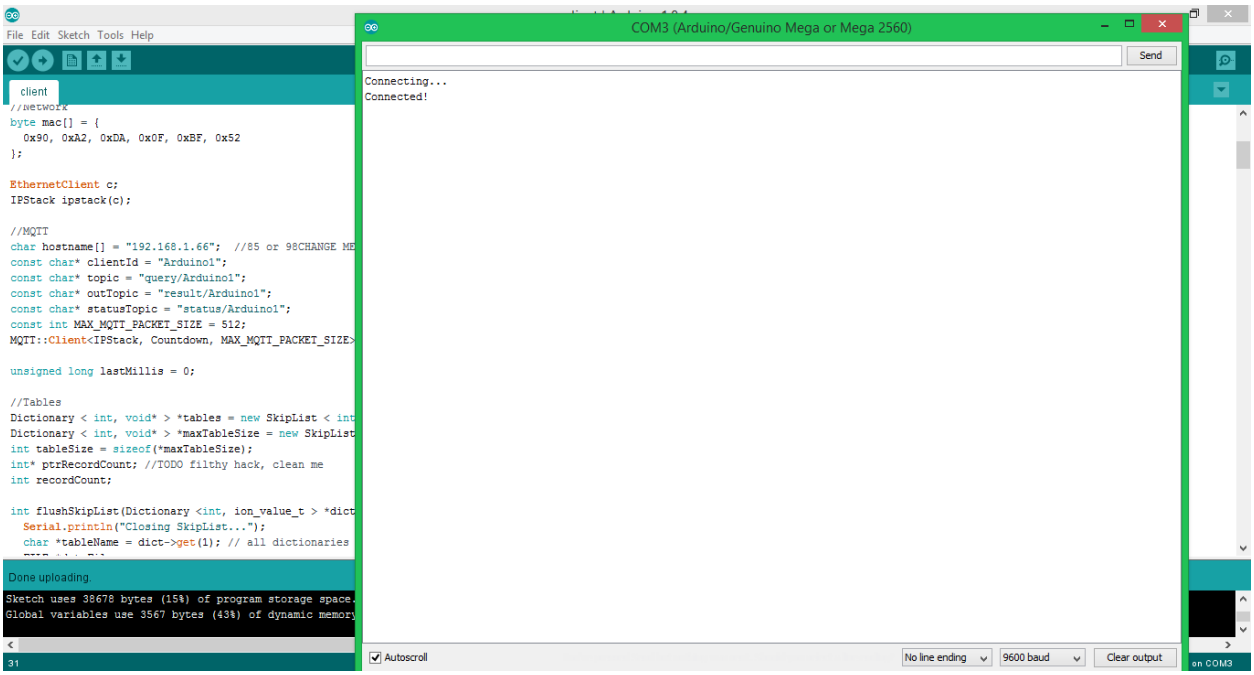

Figure 19: Serial Monitor showing the Microcontroller is connected

4.5 Web Site

Navigate to the web_portal directory and open main.html with Chrome, or Firefox, if everything is running and connected properly you should see a check box with Arduino1 by it and the top right corner should say connected. From here you can type in queries press send and view the results. As well you can export any Select query result into a CSV from the site.

When you type out a query and press send the query will disappear from the Query box and appear in the Query History box in chronological order. As well there will be a loading bar at the top that says ‘waiting for results’ so you know the application is doing something. If a messaged is returned the bar will change to say ‘message received!’ and information about it will be displayed in the Query Results box. If an error has occurred the bar will change to show the appropriate error message.

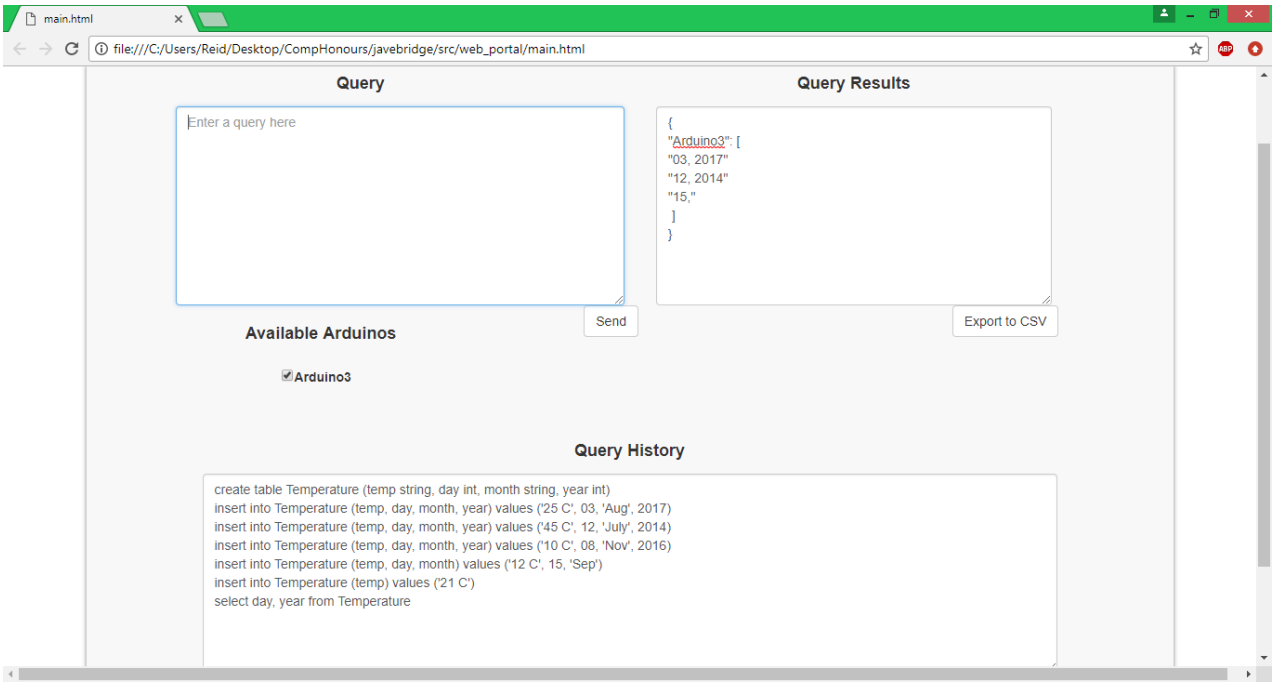

Figure 20: Users web portal after some queries have been sent to the Arduino

After a user has sent a query off to an Arduino Board through the web portal, the query along with its IonDB translation and the chosen specified Arduino Board will appear on the Server terminal as well as on the Arduino’s Serial Monitor.

```
yarn run v1.2.0
$ node server.js
Apr 14, 2018 8:00:50 PM java.util.prefs.WindowsPreferences <init>
WARNING: Could not open/create prefs root node Software\JavaSoft\Prefs at root 0x80000002. Windows RegCreateKeyEx(...) r
eturned error code 5.
temp app running at http://0.0.0.0:3000/
{ queryString: 'create table Temperature (temp string, day int, month string, year int)',
  targets: [ 'Arduino3' ] }
published {"op_code":"c","query":{"table":"Temperature","fields":"temp:s;day:i;month:s;year:i;"}}
message received
result/Arduino3 "Table created;EOR\u0000"
{ queryString: 'insert into Temperature (temp, day, month, year) values (25, 03, 08, 2017)',
  targets: [ 'Arduino3' ] }
published {"op_code":"i","query":{"table":"Temperature","fields":"25:temp;3:day;8:month;2017:year;"}}
message received
result/Arduino3 "Record inserted;EOR\u0000"
```

Figure 21: Server Terminal after a user has created a few queries for Arduino3

```
Connecting...
Connected!
{"op_code":"c","query":{"table":"Temperature","fields":"temp:s;day:i;month:s;year:i;"}}
createTable called
Finished creating table
Closing SkipList...
Sending Table created;EOR to result/Arduino3
{"op_code":"i","query":{"table":"Temperature","fields":"25:temp;3:day;8:month;2017:year;"}}
Table gotten
25:temp;3:day;8:month;2017:year;
Record inserted...
Closing SkipList...
Sending Record inserted;EOR to result/Arduino3
Finished insert
```

Figure 22: Arduino's Serial Monitor after a user has created a few queries for Arduino3

5. Future Work

As with most programming applications the vision of what Ion Field could become and accomplish were high but due to errors and unforeseen complications not everything that was planned for was completed in the time limit. For the future adding more functionality for delete queries would be helpful. Users should be able to choose to delete only certain attributes or tuples from a table based on certain restriction/conditions. Also users should be able to drop tables from the database if needed. The ability to update tables and perform update queries on tables in the database would be extremely helpful and increase the capabilities of the application.

Besides work on the query parser, the whole purpose behind this application is to create a program for a microcontroller device that can be placed outside in fields and orchards to monitor the temperature, precipitation, sunlight and other factors that may affect plants. There would be a network of Arduinos outside, each gathering its own information which would be stored locally on the Arduino board. A user should be able to view this information from chosen Arduinos using the web portal. With this in mind sensors would have to be added to the board and code created for them to monitor and store the proper information. As well the Arduino board would have to be packaged and encased properly so it can stay outside in different environmental conditions.

6. Conclusion

Overall Ion Field Network is still a work in progress, but it has advanced with much more functionality. The application capabilities have greatly increased along with its robustness. The ability to parse many different forms of statements from users and decrease the previously very strict syntax requirements has made a much more user friendly platform. The biggest breakthrough is the ability to select what information you wish to see from the database and how

you wish to view it. Users can now perform SELECT queries with multiple tables and clauses in the statement, they can group, order, and filter etc. the information. This was extremely beneficial because how a user may view the data stored in a database is very important, especially as more information is added and the tables grow. Users do not want to call up all the information and have to sort through it themselves one table at a time. That is costly and time consuming and prone to error.

Many things were learned throughout the project, and I gained lots of knowledge working on it. I had no previous understanding of microcontrollers and how they function or of how to make programs for them. The application is written with many different languages C/C++, JavaScript, HTML, CSS and Java - a few of which I was required to learn. Implementing the bridge between JavaScript and Java was something I was unaware could be accomplished before. With that I learned how to create a Java object in JavaScript and how to use their methods accordingly. I enjoyed working on Ion Field Network and hope to see it continue to grow.

Bibliography

[1] J. Jaume, "The Importance of Queries: The Key to Useful SM Data," *Brandwatch Blog,* p. 1, 2014.

[2] R. McQueen, D. Smekal, D. Olychuck and S. Macbeth, "Ion Field Network," University of British Columbia Okanagan, Kelowna, 2017.

[3] "MQTT Frequently Asked Questions," MQTT, 7 Nov 2014. [Online]. Available: http://mqtt.org/faq. [Accessed 23 Feb 2018].

[4] "Moquette MQTT," Eclips Foundation, 2017. [Online]. Available: https://projects.eclipse.org/projects/iot.moquette. [Accessed 28 Feb 2018].

[5] Wikipedia contributers, "Embedded system," Wikipedia, The Free Encyclopedia, 13 March 2018. [Online]. Available: https://en.wikipedia.org/wiki/Embedded_system. [Accessed 20 March 2018].

[6] "What is Arduino," Arduino, 2018. [Online]. Available: https://www.arduino.cc/. [Accessed 28 Feb 2018].

[7] Nantonos, Artist, *Arduino Mega2560 R3 pinouts photo.* [Art]. https://forum.arduino.cc/index.php?topic=125908.0, 2012.

[8] "UnityJDBC Data Virtualization," UnityJDBC, 2018. [Online]. Available: http://www.unityjdbc.com/. [Accessed 2 March 2018].

[9] S. Fazackerley, E. Huang, G. Douglas, R. Kudlac and R. Lawrence, "Key-Value Store Implementations for Arduino," IEEE, Halifax, 2015.

[10] R. Lawrence and G. Douglas, "LittleD: A SQL Database for Sensor Nodes and Embedded Applications," *Proceedings of the 29th Annual ACM Symposium on Applied Computing, ser. SAC '14,* pp. 827-832, 2014.

[11] N. Anciaux, L. Bouganim and P. Pucheral, "Memory Requirements for Query Execution in Highly Constrained Devices," *Proceedings 2003 VLDB Conference,* pp. 694-705, 2003.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Syst.,* vol. 30, no. 1, pp. 122-173, 2005.

[13] N. Dunkels and A. Tsiftes, "A Database in Every Sensor," *ser. SenSys'11. New York, NY, USA: ACM,* pp. 316-332, 2011.

[14] P. Bonnet, J. Gehrke and P. Seshadri, "Towards Sensor Database," Springer-Verlag, London, UK, 2001.

[15] N. Osegi and P. Enyindah, "GOptimaEmbed: A Smart SMS-SQL Database Management System for Low-Cost Microcontrollers," *CoRR,* vol. 1, 2015.