Vol.2017-OS-139 No.7
2017/3/1

Low-power Distributed NoSQL Database with Message Queue Protocol on Embedded System

PAETHONG PORNPAT[†1] MITARO NAMIKI[†1]

Abstract: The Internet will become the Internet of Things (IoT) that is able to have an immediate access to information of the physical world and its objects. These technologies allow us to achieve simpler interaction between the physical world and the virtual world by integrating a large number of real-world sensors into the Internet. This achievement provides connectivity for everyone and everything that embed some intelligence in Internet-connected objects to communicate, exchanges information, make decisions, invoke actions and provide amazing services. In past decade, many technologies; software, hardware and embedded objects are increasing. For example, a credit-card sized computer, such as Raspberry Pi. It is one of the key platforms for IoT and it is a really popular platform since it offers an entire Linux server within a small device that is very economical and provides sufficient performance. Moreover, it also provides a GPIO for directly connecting to multiple sensors. In this research, we will explain how to construct database server for embedded IoT middleware that has data distribution and low-power consumption by using credit-card sized computer and message queue, which has acceptable performances and affordable price. It is a great platform for interacting with many ubiquitous sensing devices of residential environment, such as homes, offices, or farms.

Keywords: Credit-card Sized Computer, Distributed Database, Embedded System, Internet of Things, Message Queue, MongoDB, MQTT, NoSQL Database, Low-power, Raspberry Pi

1. Introduction

Objects and Information of our world can be easily accessed nowadays resulting in the birth of the terminology, Internet of Thing (IoT) [1]. With IoT concept, it does not only help facilitate integration between the real-world devices and virtual world information, but it also covers the infrastructure, such as software, services, and hardware in order to support the physical world objects networks. A plain integration between the physical world and the virtual world can be delivered by IoT concept, integration of vast number of real-world physical devices and the internet, and this furnishes the connections between people and every embedded intelligence in Internet-connected objects. Thus, it ensures that communication, information exchange, decision making, amazing services are delivered. IoT can be counted as disruptive technology where a new ubiquitous computing and communication era emerge.

Database is considered as one of the most vital components in IoT, and its roles are gathering and reserving lots of data from ubiquitous sensing devices. The presence of smart devices can sense physical objects and interpret them into a flow of information data. Similarly, the IoT devices can trigger actions, maximize safety, enhance security, provide comfort, and furnish energy-savings. Those mentioned devices will achieve approximately 26 billion connected devices by 2020 [3]. Furthermore, it is crucial to progress artificial intelligence algorithms, which will be centralized or propagated for supporting people.

For example, the credit-card sized computer, such as Raspberry Pi. It is one of the key platforms for IoT, and it is really popular platform since it offers an entire Linux server within a small device that is very economical and provides sufficient performance. Moreover, it also provides a general-purpose input/output (GPIO) for directly connecting to multiple sensors.

2. Issues and Goals

In recent year, the Internet will become the Internet of Things (IoT) that is able to have an immediately access to information about the physical world and its objects. These technologies allow us to achieve simpler interaction between the physical world and the virtual world by integrating a large number of real-world sensor into the Internet. This achievement provides connectivity for everyone and everything to communicate, exchanges information, make decisions, invoke actions and provide amazing services.

In past decade, many technologies, gadgets, electronic devices, hardware; temperature sensors, proximity sensors, pressure sensors, water quality sensors, chemical/smoke/gas sensors, level sensors, IR sensors, endless sensing capabilities, credit-card sized computer, and embedded module (GPIO, WiFi module) and software, such as NoSQL database for data analysis and visualization were built to support Internet of Things (IoT). A new protocol for communication between a hardware using extremely less energy is motivation that makes people design IoT system. Many IoT technologies are based on open source, but some of them are integrated with cloud computing technology for collecting, storing, and processing the massive amount of data. The consequences of these are storages used to contain these data.

This research will illustrate how to structure database with message queue distribution mechanism for IoT middleware that has data distribution with message queue on credit-card sized computer like Raspberry Pi. To clarify, Raspberry Pi has acceptable performances and economical price. It is great platform for interacting with many ubiquitous sensing devices of residential environment, such as home, office, or farm.

†1 Tokyo University of Agriculture and Technology

3. System Architecture

The system contains three main components: master, data nodes, and metadata. Master and data node work together as a single system; the master node is able to connect to other data nodes and vice versa. Metadata table as shown in Figure 1.

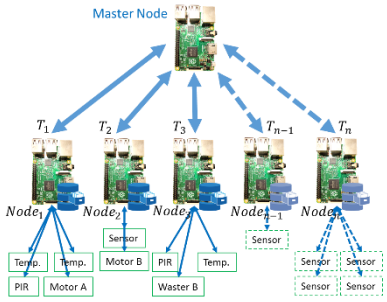

Figure 1: System Structure and Design

Every node, each node runs NoSQL database as a server, works completely independent from each other to store and process data from connected sensor hardware. Moreover, every node can be a master node when clients request to connect to that node, and other node will be a data node for providing any information.

For data distribution, each node will receive data and store it into its database. After that, it will broadcast an updated data message to the other node by using message queue protocol. In this case, it means all database operation including add, update, delete events fires a broadcast updated data message to the other node in the system. With this reason, each node has a metadata table as a private data in its database, which contains information about the node's data such as IP address, namespace, last active, and etc.

3.1 Software Components

The components of software are contained with MongoDB database and MQTT client for message queue communication. They are running on operating systems such as Linux-base. The MQTT client is embedded in MongoDB database engine as shown in the Figure 2.

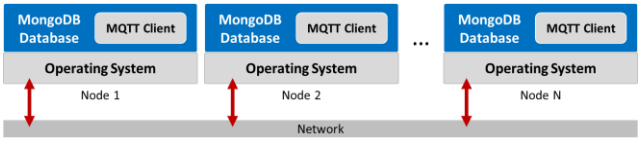

Figure 2: Software Architecture

3.1.1 MongoDB Database

The MongoDB Database is adopted to this system because it is the one of the most popular databases for IoT in the world since it has capability to store any kind of data, analyze it in real time, and change the schema as the business go. With the document model of MongoDB, it enables us to store and process data of any structure: events, time series data, geospatial coordinates, text, and binary data. We can adapt the structure of document's schema just by adding new fields making it simple to handle the rapidly changing data generated by IoT applications. With multiple options for scaling including range-based, hash-based and location-aware sharding, MongoDB can support thousands of nodes, petabytes of data, and hundreds of thousands of ops per second without requiring us to build custom partitioning and caching layers.

3.1.2 MQTT Broker (mosquitto)

Eclipse Mosquitto™ is an open source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for "Internet of Things" messaging such as low power sensors or mobile devices: phones, embedded computers, or microcontrollers like the Arduino.

3.1.3 Embedded MQTT C/C++ Client Libraries

This is an embedded MQTT client library for C/C++. It was written with Linux and Windows in mind. Also, it assumes the existence of Posix or Windows libraries for networking (sockets), threads and memory allocation. The embedded libraries are intended to have these characteristics:

- Use very limited resources - pick and choose the needed components
- Not reliant on any particular libraries for networking, threading or memory management
- ANSI standard C for maximum portability, at the lowest level
- Optional higher layers in C and/or C++

3.2 Hardware Components

It contains master node and data nodes, which work together as a single system mentioned earlier. It also contains some connected sensor.

3.2.1 Master Node and Data Node

This system is able to use any credit card-sized single-board computers such as Rasberry Pi, Banana Pi, and etc. For a hardware platform, it should be able to run a Linux-base operating system or other operating systems, and it requires to run a MongoDB database also.

Table 1: Minimum Requirements:

CPU	900MHz
Memory	1GB
Storage	Over 2 GB
OS	Linux-base Operating System (Raspbian)
GPIO	Yes
Network	Yes

3.2.2 Sensors

Regrading to this system connected to any sensor, a sensor is one of the most important components. However, we can use any sensors that are able to connect with credit card-sized computers.

Mostly, a general sensor fully support such as temperature sensors, proximity sensors, pressure sensors, water quality sensors, chemical/smoke/gas sensors, level sensors, IR sensors, endless sensing and so on.

4. System Design

In recent years, data collection from many IoT sensors becoming more challenging and important. These data can benefit society in many ways. Thus, the efficiency and performance of storing these data from hardware devices (such as many types of sensor) are main purposes of this research. We are promising a solution of low-power distributed database on credit-card size computer as a node. All node is running NoSQL database, and they are completely independent from each other. Besides, it will communicate for data exchange by using message queue.

4.1 Master Node

It acts as a proxy server to communicate with client to access database. Firstly, the master node receives query command from the client to process and search through what is the client requires. Next, master node will take required data from client to search in metadata table. Metadata table will show where is the required data in data node. Thirdly, the master node will forward the query command and input from client to each data node for processing. Then, the result from all of data nodes will be sent back to master node. Finally, the master node will combine all of data together and respond back to the client as show in Figure 4(a).

4.2 Data Node

It is a device that collects and stores gathered data from sensors. In the normal state, data node always collects data from sensors periodically. However, when requested data command come from master node, the data node will process client's command and respond result back to the master node as show in Figure 4 (b).

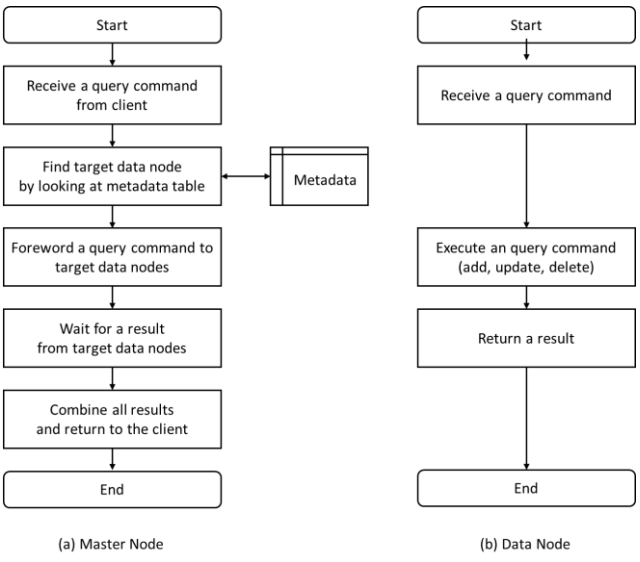

Figure 4: Master Node and Data Node Flowchart Diagram

4.3 Metadata Table

As an individual working, all node will manage all collection by itself. It means a real data is stored on this node. However, it has to describe in metadata on each node about its data.

Metadata Structure:

```
{
   ns: "sensers.type",
   key: "temp"
   nodes: {
       "192.168.1:2017",
       "192.168.5:2017",
       "192.168.100:2018"
   },
   last_active: "2016-10-18T18:25:43.511Z",
   active_duration: "60"
}
```

Table 3: Metadata Structure Description

Name	Description
ns	(string) The string of collection and field
key	(string) The key of data in data node.
nodes	(array of string) The list of server id, server name, server IP address of data nodes.
last_active	(date) The timestamp of last active of data node.
active_duration	(int) The duration number of active time in second of data node. For example; 60 sec, 120 sec, etc.

Data Document Structure:

```
{
   _id: ObjectId(…),
   key: "temp",
    ...
}
```

Table 3: Data Document Structure Description

Name	Description
_id	(string) This an original id of MongoDB
key	(string) The key of data in data node.
…	(N/A) Optional data that we would like to added in a document.

4.4 Distribution Mechanism

In short, distribution is a process of distributed data that will be stored in multiple computers locating in the same physical location; or may be dispersed over a network of interconnected computers. We can distribute collections of data (e.g. in a database) across multiple physical locations. A distributed database can reside on organized network servers or decentralized independent computers on the Internet, corporate intranets, extranets, or other organization networks. Because they store data across multiple computers, distributed databases may improve performance at end-user worksites by allowing

transactions to be processed on many machines, instead of being limited to one machine.

Regarding to our system architecture, all nodes collect data from connected sensors. When each node work independently, the data from each node are not synchronized. This problem can be solved by using metadata synchronization instead of using normal data distribution method of MongoDB sharding as shown in Figure 3.

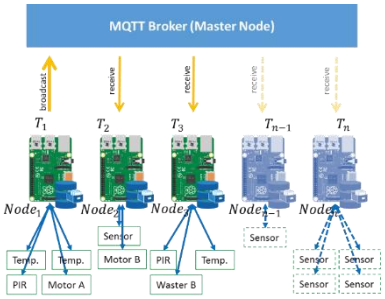

Figure 3: Metadata with MQTT Message Exchanging

In this metadata synchronization method, each data node is required to have metadata table in its database. The metadata table contains IP address, namespace, data key, last active for reference what data are contained in other data nodes. When a node has been added, updated, or deleted, this node will broadcast its updated data to other nodes by using message queue protocol (such as MQTT). This metadata synchronization method allows data nodes to access other another data nodes with reduction in nodes power consumption as shown in Figure 4 and 5.

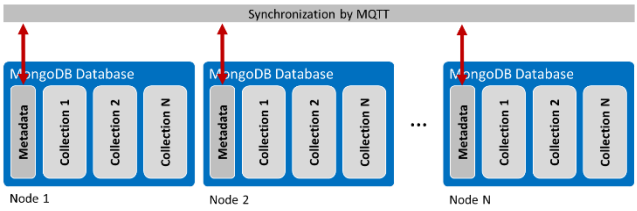

Figure 4: Metadata Table in Database

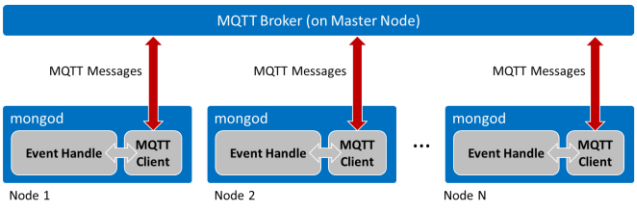

Figure 5: Add Event Handler and MQTT in MongoDB Process

4.5 Scenario

To get better understanding, we will explain about our distributed database by using message queue. In Figure 6.7, there are N nodes that collect and store data from many types of sensors in many places. All nodes are placed in different locations. A case in point, the Node #1 is planted in bathroom, Node #2 is placed in the garden, Node #3 is installed in the kitchen and other nodes are installed in many different locations. Node #1, it is connected to temperature sensors, passive infrared sensor (PIR sensor), and motor sensor. The two temperature sensors are sending temperature data every period of time. The 1st temperature sensor send its temperature data every five minutes, and 2nd temperature sensor send its temperature data every ten minutes. For Node #2, it is connected to motor sensor, which sends back status of sensors, quantity of speed and velocity. For Node #3, it is connected to PIR sensor, which sends back its status. Next, waterflow sensor is a device that use to measure to amount of water and send back to the node. The last one is temperature sensor, which is the same type of sensor in Node #1. For other nodes, they are connected to other type of sensors and send the data back to their own node to store it as show in Figure 6.

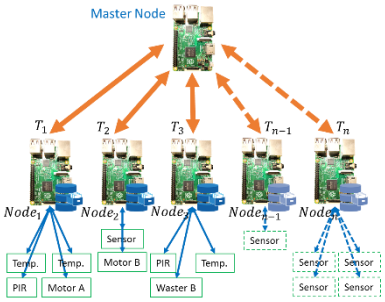

Figure 6: Mater Node and Data Nodes

When the master node get query command from client for getting all temperature data, it will process the command and look up at a metadata table for finding IP address of a node that contains temperature data. After looking up from metadata table, the master node will get targeted nodes that have temperature data and request query command from the client as shown in Figure 7.

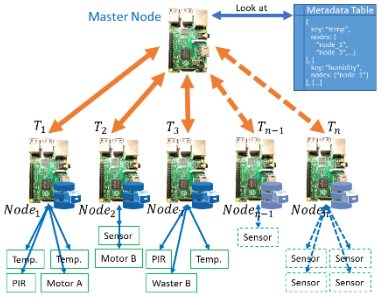

Figure 7: Mater Node Looking Metadata Table

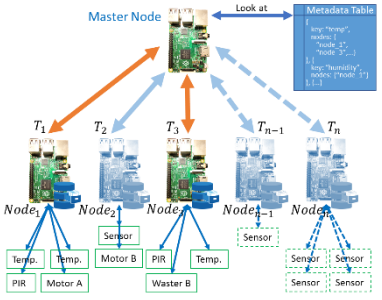

Figure 8: Mater Node Connected to Target Data Node

When the master node knows the targeted IP address of data nodes that have temperature data from the metadata table. The master node connects to data nodes as a parallel process and forward a query command from client to data nodes. Then, each

data node executes that query command for getting its temperature data that is collected from connected sensors, and it will return a result back to the master node. A master node will wait until all results from each data node are completed. Finally, the master node will consolidate and combine the temperature result, and send back to the client as Figure 8.

5. Evaluation

We evaluate the proposed method. The MongoDB database as individual running by itself on Raspberry Pi are evaluated and compared with a MongoDB database by sharding function database operation performance.

5.1 Experimental Study of Power Consumption

We will describe about our experimental study on MongoDB on x86 machine and MongoDB on credit card-sized single-board computers. In environment of experimental study, we used a MongoDB version 2.6.0 on x86 machine of Inter Core i7 CPU 870 @ 2.93 GHz with 4 cores, 3.9 GB of memory, 128 GB of HDD storage, and Ubuntu 14.04 LTS 64-bit of operating system, And MongoDB version 2.4.10 for Raspberry Pi 2 Model B of 900MHz quad-core ARM Cortex-A7 CPU, 1 GB of memory, 16GB of SD card as local-base storage and Raspbian for operating system.

5.1.1 Raspberry Pi 2 Power Consumption Environment

First of all, we would like to know about power consumption of Raspberry Pi in normal state and other state. By using digital multimeter tool, it measures the power consumption in milliampere (mA), and we calculate it in watt (W) at fixed voltage (5 voltages for Raspberry Pi) following this equation: (Watts = Amps x Volts) and ApacheBench (ab), it is a very handy web server benchmarking tool as shown table 4 below.

Table 4: Raspberry Pi 2 Power Consumption

Raspberry Pi 2	**State**	**Power Consumption**
Model B	Power Off (plugged in)	20-30 mA (0.1W)
Model B	Idle	280-420 mA (1.5W)
Model B	ab -n 100 -c 10 (uncached)	900-1200 mA (~4.5W)
Model B+	Power Off (plugged in)	20-30 mA (0.1W)
Model B+	Idle	230-240 mA (1W)
Model B+	ab -n 100 -c 10 (uncached)	480-800 mA (~2.4W)

Table 4 shows the statistics of power consumption in some state of Raspberry Pi 2. In power-off state, both of them are use 20-30 mA. In idle state, Model B has two times as much power consumption as Model B+. Finally, in ab test state, Model B also has higher power consumptions than Model B+.

5.1.2 MongoDB Non-sharding vs Sharding Performance

We evaluated the MongoDB database as an individual running by itself on a Raspberry Pi and MongoDB database by sharding function on four Raspberry Pis; one for router and config server, and three shard servers for database performance and energy usage by using the “Sysbench Benchmark” (https://github.com/tmcallaghan/sysbench-mongodb) for MongoDB and TokuMX. In the default configuration, the benchmark creates sixteen collections, each with ten-million documents for benchmarking a database performance; inserting time, insert per second (IPS), online transaction time, and transactions per second (TPS).

Table 5: Database Performance Comparison

Database Performance	**MongoDB (Individual Running)**	**MongoDB Sharding on 4 x Raspberry Pi**
Inserting Time	475 seconds	4,875 seconds
Insert per Second (IPS)	3,363.17	328.14
Online Transaction Time	580 seconds	600 seconds
Transactions per Second (TPS)	12.83	11.74

Table 5 shows a result of database performance of MongoDB individual running on one Raspberry Pi compared with MongoDB sharding on four Raspberry Pi. It was faster than MongoDB sharding, and inserted per second (IPS) performance was also higher than MongoDB sharding. For online transaction and transactions per second (TPS) performances, they have almost the same value.

In addition, when started running a benchmark program on above, we also evaluated an energy usage on both of them by using a digital multimeter and calculate it in joule (J) as shown in table 3.

Table 6: Energy Usage Comparison

Energy	**MongoDB (Individual Running)**	**MongoDB Sharding on 4 x Raspberry Pi**
Idle mA	250 mA (1,000 mA)	1,000 mA
Avg. Execution mA	300 mA (1,200 mA)	1,250 mA
Inserting Energy (J)	142.5 kJ (570 kJ)	6,093.75 kJ
Transaction Energy (J)	174 kJ (700 kJ)	750 kJ

Table 6 shows results of energy usage of individual MongoDB running on one Raspberry Pi compared with MongoDB sharding on four Raspberry Pi. In an idle state, both of them used 1,000 milliamps and MongoDB sharding use over 50 milliamps in average execution.

5.2 Inserting Performance

We evaluated inserting performance of MongoDB with our implementation compared to MongoDB sharding by preparing five number of nodes (one node, two nodes, three nodes, four nodes, five nodes) to insert as one, five, ten, fifty, one-hundred, five-hundred, and one-thousands of data records.

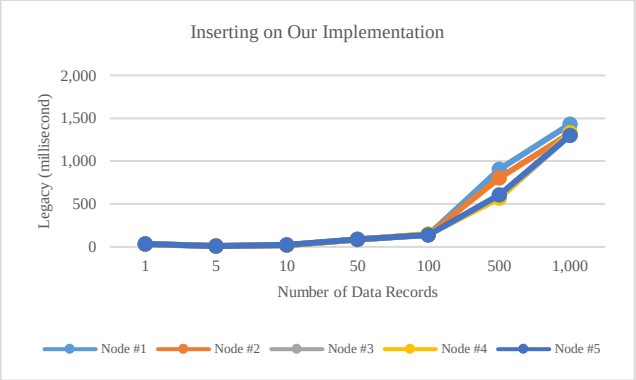

Figure 9: Inserting Performance on Our Design

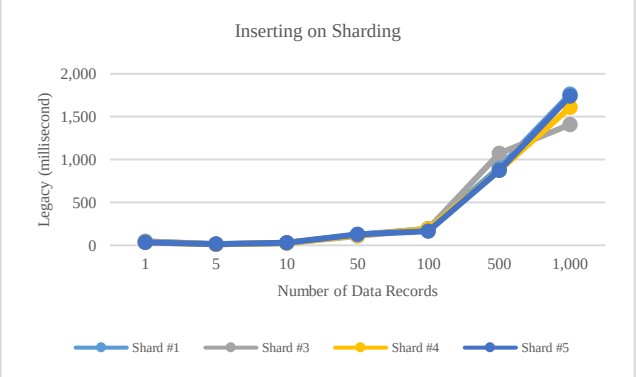

Figure 10: Inserting Performance on Sharding

In the Figure 9 and 10, the graph shows the result of inserting performance compared between MongoDB with our implementation and MongoDB sharding. The number of record data in one record, five records, ten records and fifty records in our implementation and MongoDB sharding show that the amount of time consumption is similar in both methods. However, when it came to large size of data as one-hundred records, five-hundred records and one thousand, the time required for inserting data between two method are different from each other. Our implementation shows that in big data our performance is better that MongoDB sharding. In other hand, the result[results] shown[show] the small [number] of record data the amount of time to insert data is similar to each other; however, in the large record data the amount of time that use for insert the data is different significantly.

5.3 Updating Performance

We evaluate a performance of data updating of MongoDB with our implementation compared to MongoDB sharding by preparing five number of nodes (one node, two nodes, three nodes, four nodes, five nodes) that contain one thousand, five thousand, ten thousand, fifty thousand, one-hundred thousand, five-hundred thousand, one million of data records.

5.3.1 Data Updating Without Index Condition

In the Figure 11 and 12, the graphs below show the result updating without indexing performance compared between MongoDB with our implementation and MongoDB sharding. The number of record data in one record, five records, ten records and fifty records in our implementation and MongoDB sharding show that the amount of time consuming is similar in both methods.

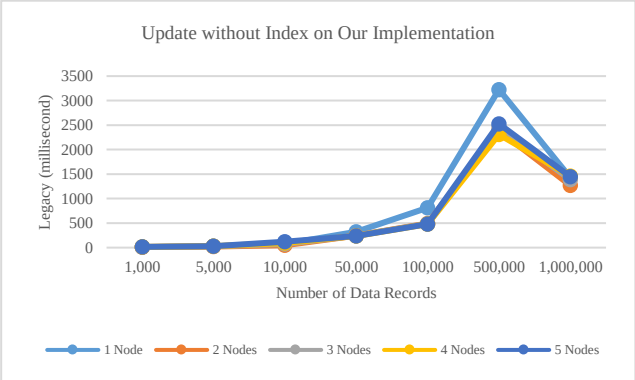

Figure 11: Updating without Indexing Performance on Our Implementation

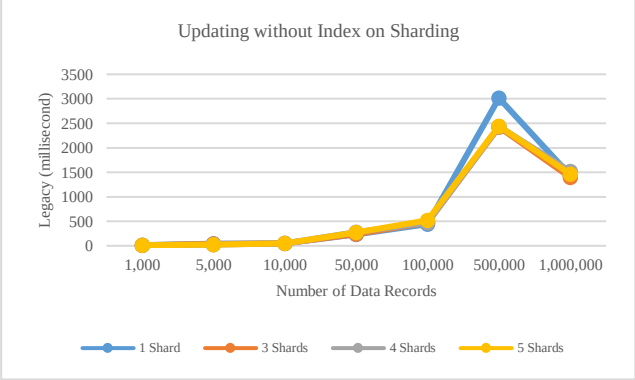

Figure 12: Updating without Indexing Performance on Sharding

The result was good because our implementation and sharding are almost similar performance but it used less energy. Only the one node of our implementation, it is a little slower than sharding when it has to updating a data without index condition in very large number of data record (more than 100,000 record of data).

5.3.2 Data Updating Using Index Condition

In the Figure 13 and 14, the graph show the result updating by using indexing performance comparing between MongoDB with message queue by our implementation and MongoDB sharding.

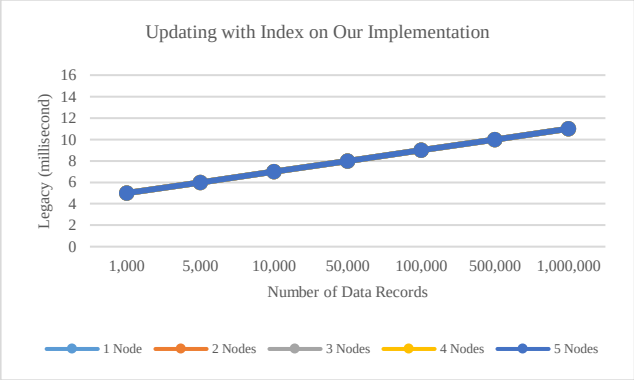

Figure 13: Updating with Indexing Performance on Our Implementation

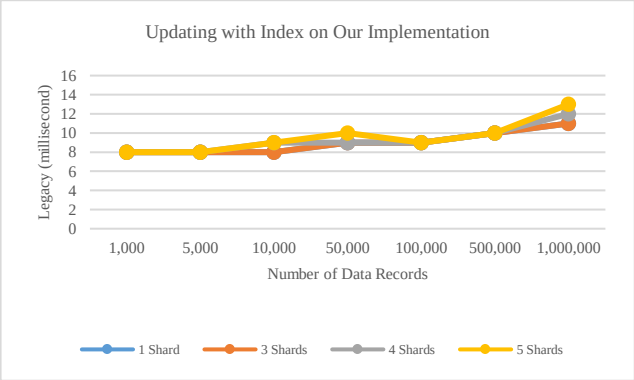

Figure 14: Updating with Indexing Performance on Sharding

The result was very good because MongoDB with our implementation was faster than sharding when the number of data is not too high. However, it was similar performance with sharding when the number of data is going up. Moreover, our implementation still has low power compulsion than sharding.

5.4 Deleting Performance

We evaluated a performance of data deleting of MongoDB with our implementation compared to MongoDB sharding by preparing five number of nodes (one node, two nodes, three nodes, four nodes, five nodes) that contain one thousand, five thousand, ten thousand, fifty thousand, one-hundred thousand, five-hundred thousand, one million of data records.

5.4.1 Data Deleting Without Index Condition

In the Figure 15 and 16, the graph show the number of record data in one record, five records, ten records and fifty records in our implementation and MongoDB sharding show that the amount of time consuming is similar in both methods.

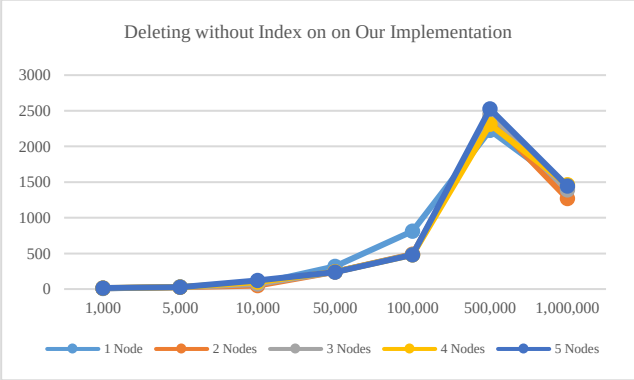

Figure 15: Deleting without Indexing Performance on Our Implementation

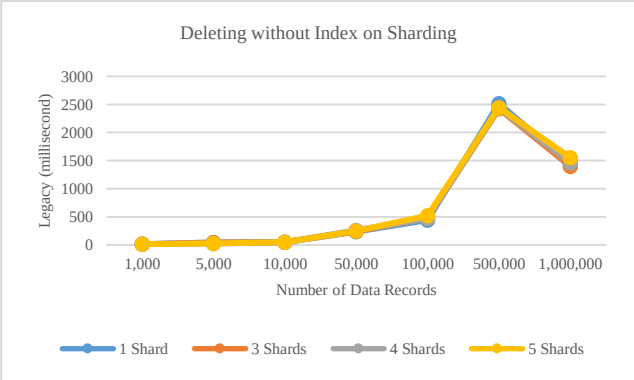

Figure 16: Deleting without Indexing Performance on Sharding

In term of performance, the result was not good but not bad because MongoDB with our implementation and sharding are almost similar performance for deleting without index. However, our implementation still has low power compulsion.

5.4.2 Data Deleting Using Index Condition

In the Figure 17 and 18, the graph above show the number of record data in one record, five records, ten records and fifty records in our implementation and MongoDB sharding show that the amount of time consuming is similar in both methods.

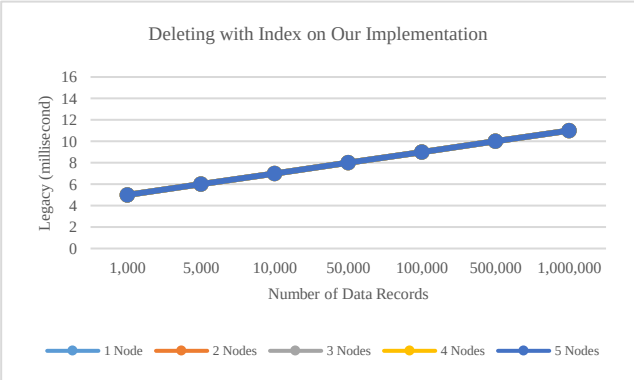

Figure 17: Deleting with Indexing Performance on Our Implementation

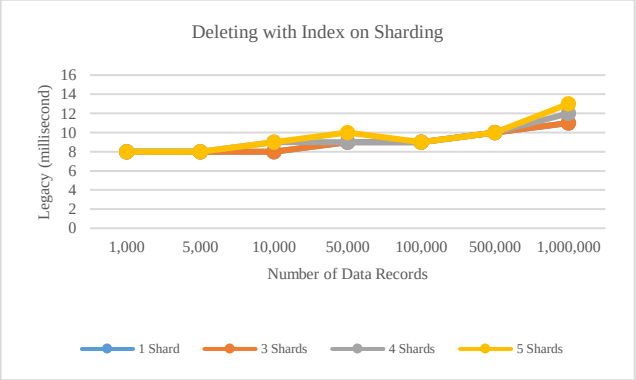

Figure 18: Deleting with Indexing Performance on Sharding

In term of performance, the result was not good but not bad either because MongoDB, with our implementation and sharding, is almost similar, in terms of performance for deleting without index. However, our implementation still has low power consumption.

6. Discussion

As an evaluation result, we mainly use Raspberry Pi Model B for each evaluation. Its power consumption rate is around 280~420 milliampere (mA) or ~1.4 watts (W) in idle state and 380~480 mA (~2W) in working state for database execution.

Then, we evaluated the database performance by comparing between MongoDB non-sharding and sharding. The result was very interesting because MongoDB with non-sharding used has less power consumption than MongoDB with sharding. In my opinion, all sharding node have to communicate with each node all the time. In this point, it inspired us to run a MongoDB as individual database that collect and store a data from the sensors that are connected and using MQTT for asynchronous communication for data distribution with low-power. A metadata table was adopted into each database for data synchronization.

In inserting performance, the MongoDB with message queue protocol by our implementation is similar to MongoDB with sharding function when the number of data is small but it will be different when the number of data is bigger; the MongoDB with our implementing is better than MongoDB sharding because all node of our implementation is run as individual node. In updating and deleting performance, both of them has almost similar performance. However, MongoDB with our implementing was used has less power consumption as the pervious reason above.

7. Conclusion

To recapitulate my analysis, the construction database server for IoT that has data distribution and low-power consumption by using credit-card size computer like Raspberry Pi which have acceptable performances and affordable price. It is perfect platform for interacting with many ubiquitous sensing devices of residential environment such as home, office, or farm.

Moreover, the MongoDB with message queue by our method implementing compare to MongoDB sharding on credit-card size computer. It shown MongoDB with our method implementing is quick faster than MongoDB sharding. The benefit difference is MongoDB with our method implementing used more less power consumption because all node is work independently. It has good inserting performance, updating performance, and deleting performance. It achieved our goals for data distribution with low-power on embedded system.

8. Future Work

As our implementation was focus on database as individual running and used MQTT for metadata distribution. As this point, we would like to make this system as no single-point of failure and improve a performance of metadata synchronization.

8.1 No single-point of failure (SPOF)

Since, we are currently using MQTT Broker only for Master Node, if the Master Node were to stopped working, all the other nodes can still collect data from sensors but will not be able to exchange any data.

8.2 Improve a performance of metadata

We are using various data of nodes for designing, for example, AB. But if a parameter were to be added into the rules of Data Node searching, it will hasten the process in which the data can be recalled.

References

[1] Gubbi, Jayavardhana, et al. “Internet of Things (IoT): A vision, architectural elements, and future directions.” Future Generation Computer Systems, Vol.29 No.7, pp.1645-1660, 2014.

[2] Zarghami, Shirin. “Middleware for Internet of things.”, Master Thesis of Faculty of Electrical Engineering, Mathematics and Computer Science Software Engineering, University of Twente, 2013.

[3] Kelly, Sean Dieter Tebje, Nagender Kumar Suryadevara, and Subhas Chandra Mukhopadhyay. “Towards the implementation of IoT for environmental condition monitoring in homes.” IEEE Sensors Journal, Vol.13 No.10 pp.3846-3853, 2013.

[4] Anwaar, Waqas, and Munam Ali Shah. “Energy Efficient Computing: A Comparison of Raspberry PI with Modern Devices.” Energy Vol.4 No.2, 2015.

[5] Snyder, Robin M. “Power monitoring using the Raspberry Pi.” 46th Annual Conference Proceedings of the 2014 ASCUE Summer Conference, pp.82, 2014.

[6] Aminu Baba, Murtala, “Design and Implementation of a Home Automated System based on Arduino, Zigbee and Android”, International Journal of Computer Applications, Vol.97 No.9, pp.37-42, 2014.

[7] Parker, Zachary, Scott Poe, and Susan V. Vrbsky. “Comparing NoSQL MongoDB to an SQL DB.” Proceedings of the 51st ACM Southeast Conference. ACM, 2013.