A picture containing text, circle, screenshot, graphics

Description automatically generated

2023

Exploring NoSQL Databases: A Comprehensive Analysis and Implementation for IoThings

[Company name] | [Company address]

Modern data Stores report

Dharmarlou Bowen - 20123076

Contents

[**Introduction** 4](#_Toc135040451)

[**Section A – Theoretical** 4](#_Toc135040452)

[Relational Databases & NoSQL 4](#_Toc135040453)

[Comparison of Paradigms 4](#_Toc135040454)

[Strengths and Weaknesses of NoSQL 5](#_Toc135040455)

[Types of No SQL (Key.) 6](#_Toc135040456)

[Key-Value Based 6](#_Toc135040457)

[Graph-Based 6](#_Toc135040458)

[Column-Oriented 7](#_Toc135040459)

[Document-Based 7](#_Toc135040460)

[NoSQL Database Formats 7](#_Toc135040461)

[MongoDB 7](#_Toc135040462)

[HBase 8](#_Toc135040463)

[Neo4j 8](#_Toc135040464)

[**Section B – Design & Implementation of Database** 9](#_Toc135040465)

[Problem Identification 9](#_Toc135040466)

[Modelling and Designing Process 10](#_Toc135040467)

[Relevant Software and Tools 12](#_Toc135040468)

[Architecture of System 12](#_Toc135040469)

[PostMan Utilization 15](#_Toc135040470)

[Conclusion 17](#_Toc135040471)

[References 18](#_Toc135040472)

[Figure 1: Design Process Flowchart 10](#_Toc135040473)

[Figure 2: Devices Table of Attributes 11](#_Toc135040474)

[Figure 3: app.js part-1 12](#_Toc135040475)

[Figure 4: app.js part-2 12](#_Toc135040476)

[Figure 5: app.js part-3 13](#_Toc135040477)

[Figure 6: app.js part-4 13](#_Toc135040478)

[Figure 7: app.js part-5 13](#_Toc135040479)

[Figure 8: db.js for mongodb connection 14](#_Toc135040480)

[Figure 9: nodemon running on port 3000 14](#_Toc135040481)

[Figure 10: Creation Op via Postman 15](#_Toc135040482)

[Figure 11: Reading Op via Postman-1 15](#_Toc135040483)

[Figure 12: Reading Op via Postman-2 16](#_Toc135040484)

[Figure 13: Updating Op via Postman 16](#_Toc135040485)

[Figure 14: Deletion Op via Postman 17](#_Toc135040486)

[Table 1: Differences of SQL vs NoSQL 4](#_Toc135040487)

# **Introduction**

The purpose of this report is to dive into the realm of NoSQL databases. We will explore the viability of them as an appropriate alternative solution to the traditional use of relational databases for use in data management. The report will be divided into two key sections. Section A (Theoretical) will be concerned with the theoretical understanding of how NoSQL Databases function as how they actually differ from their counterpart, Section B (Design & Implementation of a Database) is concerned with the planning and practical implementation of a NoSQL database centred around an open-source dataset that will be used for demonstrative purposes of how a database is setup and how it can interact with data.

# **Section A – Theoretical**

## Relational Databases & NoSQL

In the world of data management and storage there are two types of databases, SQL (Structured Query Language) and NoSQL. SQL databases follow a predefined schema that utilizes tables that are a combination of fixed columns and rows that are used to organize data in a more structured format.

Queries are the process in SQL databases to deal with data interactions On the other hand, NoSQL offers schema-less data structures. This means that they have the ability for flexible and dynamic data models. They are able to use a variety of data models such as key-value, document, graph, or column-based for example. By being able to do this it enables them to efficiently handle unstructured or semi-structured data or in some cases data that is rapidly changing.

They prioritize factors such as scalability, horizontal distribution, and high-performance data retrieval, but this can result in the sacrificing of complex transactions and the strict consistency guarantees that are generally associated with SQL databases.

### Comparison of Paradigms

The table below provide a different a summary of some of the differences between the data storage paradigms.

Table : Differences of SQL vs NoSQL

|  |  |  |
| --- | --- | --- |
|  | **NoSQL** | **SQL** |
| Data Model | Flexible and dynamic, various data models (key-value, document, graph, column-based) | Structured with predefined schema, tables with fixed columns and rows |
| Schema | Schema-less | Predefined schema |
| Scalability | Highly scalable, designed for horizontal scalability and distributed environments | Scalability challenges in distributed environments |
| Data Structure | Supports unstructured, semi-structured, and rapidly changing data | Structured data with fixed relationships and constraints |
| Transactions | Limited support for complex transactions and ACID (Atomicity, Consistency, Isolation, Durability) properties | Strong support for complex transactions and ACID properties |
| Consistency | Eventual consistency model, flexible trade-off between consistency and availability | Strong consistency guarantees |
| Query Language | Various query languages specific to each NoSQL database | SQL (Structured Query Language) |
| Use Cases | Big data, real-time applications, agile development, handling diverse data types | Traditional applications, relational data, complex transactions |
| Examples | MongoDB, Cassandra, Redis, Neo4j | MySQL, Oracle, PostgreSQL |

### Strengths and Weaknesses of NoSQL

**Strengths of NoSQL,** The capacity to scale horizontally and manage immense data volumes effortlessly are among the most significant qualities of NoSQL systems. They're devised in such a way so as to apportion information across various servers or clusters, ensuring uninterrupted growth whenever there's an increase in data demands. This flexibility allows for high-efficiency retrieval and handling of colossal datasets; ergo, it makes NoSQL databases ideal candidates for applications necessitating instantaneous responsiveness when processing large amounts of information.

Flexibility and Agility: NoSQL databases excel in handling diverse and evolving data structures. Given their attribute of being without a schema, these data systems allow for the latitude to process and stockpile unstructured or semi-structured information sans prior definitions. This flexibility allows developers to adapt the data model on-the-fly, making NoSQL databases ideal for agile development environments where requirements change frequently.

**Weaknesses of NoSQL**, Limited Query Capabilities: NoSQL databases typically provide simplified query languages or APIs compared to the rich querying capabilities of SQL databases. The trade-off for scalability and flexibility is often a reduced ability to perform complex joins, aggregations, and ad-hoc queries. NoSQL databases prioritize high-performance data retrieval based on specific access patterns but may lack the expressive power of SQL for intricate data analysis. This limitation can pose challenges when dealing with complex data relationships or when extensive querying and reporting functionalities are required.

Weaker Data Consistency Guarantees: NoSQL databases often adopt an eventual consistency model rather than strong consistency, which means that data updates may take some time to propagate across all replicas or nodes. While this approach allows for high availability and fault tolerance, it introduces the possibility of reading stale or inconsistent data during transient periods..

## Types of No SQL (Key.)

Below is discussed the different types of NoSQL databases that exists as they can be implemented in many forms.

### Key-Value Based

NoSQL databases that rely on key-value pairs are designed to store data in such a way where each unique value is tied to its corresponding and likewise unique key. The present design exudes an air of ease and efficiency, rendering it aptly geared for employment in situations necessitating expeditious data retrieval along with storage. Key-Value databases excel in scenarios such as caching, session management, and user profiles, where quick access to specific pieces of data is crucial.

### Graph-Based

Graph databases are designed to represent and store data as nodes and edges, where nodes represent entities and edges depict relationships between nodes. This model enables the efficient traversal of relationships and complex graph operations, making graph databases suitable for use cases involving highly connected and interrelated data, such as social networks, recommendation engines, and fraud detection. Graph databases provide powerful query languages and algorithms for analysing and navigating graph structures.

### Column-Oriented

This data model allows for efficient retrieval of specific columns or subsets of columns, enabling fast data aggregation and analysis. Column-oriented databases excel at handling complex queries and aggregations across massive datasets, making them ideal for applications like business intelligence, data warehousing, and time-series analysis. However, they may not perform as well for transactional workloads or scenarios that require frequent updates or inserts due to the overhead involved in maintaining columnar storage structures.

### Document-Based

Document databases store data in flexible and self-describing document formats, such as JSON or XML. Each document contains key-value pairs or key-array pairs, allowing for hierarchical and nested data structures. Document-based NoSQL databases provide schema flexibility, enabling easy adaptation to changing data requirements. They are commonly used in content management systems, e-commerce applications, and real-time collaboration tools. Document databases support powerful querying and indexing capabilities, making it easier to retrieve and manipulate document-based data. However, complex joins and transactions involving multiple documents can be more challenging compared to relational databases.

## NoSQL Database Formats

### MongoDB

**MongoDB architecture** follows a distributed architecture that allows for horizontal scaling and high availability. It employs a document-oriented data model, where data is stored in flexible, self-describing documents in BSON (Binary JSON) format. MongoDB uses a replica set configuration, where multiple copies of data are maintained across different nodes for redundancy and fault tolerance. The architecture supports automatic failover and data replication, ensuring data durability and continuous availability.

**MongoDB Schema Design Model** adopts a schema-less or flexible schema design model, allowing documents within a collection to have different structures. This flexibility enables developers to adapt the data model as application needs evolve without requiring schema migrations. MongoDB's schema design focuses on denormalization, aiming for efficient data access and reducing the need for complex joins. It supports embedded documents and arrays, enabling the representation of complex relationships within a single document.

**MongoDB Query Model** offers a rich and powerful query model that includes a flexible query language and comprehensive indexing capabilities. It supports a wide range of query operations, including equality, range, and text searches, as well as aggregation and geospatial queries. MongoDB's query language allows for complex filtering, sorting, and projection of data.

### HBase

**HBase Architecture** is built on top of the Hadoop Distributed File System (HDFS) and follows a distributed architecture designed for scalability and fault tolerance. It is a column-oriented NoSQL database that organizes data into tables consisting of rows and columns. HBase employs a master-slave architecture, where the HMaster node manages metadata and coordinates operations, while multiple HRegionServers store and serve data. The data in HBase is horizontally partitioned into regions, which are distributed across the cluster.

**HBase Schema Design Model** follows a wide-column schema design model, similar to the column-oriented databases. It allows for flexible schema design, where each row can have a varying number of columns, and columns can be added or modified dynamically. HBase tables are organized into column families, which group related columns together.

HBase (**Query Model**) provides a simple and efficient query model that supports fast key-based lookups and range scans. The primary access method in HBase is through row keys, which enable direct retrieval of individual rows. HBase also supports secondary indexes using Apache HBase Coprocessors, allowing for more complex query capabilities. However, unlike relational databases, HBase does not provide a built-in query language for ad-hoc querying. Instead, it offers the ability to perform low-level operations like get, put, scan, and delete using client APIs or HBase shell commands. To leverage HBase effectively, developers often need to integrate HBase with other query engines or data processing frameworks, such as Apache Hive or Apache Spark.

### Neo4j

**Neo4j Architecture** is a graph database that employs a native graph storage model, making it ideal for managing highly connected data. Its architecture is built around nodes and relationships, where nodes represent entities or data points, and relationships define connections or associations between nodes.

**Neo4j Schema Design Model** employs a flexible schema design model that allows for dynamic and evolving graph structures. It does not enforce a rigid schema like traditional relational databases. Instead, Neo4j allows nodes and relationships to have different properties and relationships, providing flexibility in representing complex, evolving domains..

**Neo4j Query Model** provides a powerful and expressive query language called Cypher, specifically designed for querying graph data. Cypher offers a declarative syntax that allows developers to express complex graph patterns and relationships in a concise and intuitive way. With Cypher, users can traverse the graph, filter and match nodes based on properties, and perform aggregations and calculations on graph data.

# **Section B – Design & Implementation of Database**

* Focus on designing the sensors database for IoThings' extended computer systems.
* Build a NodeJS server to handle MQTT messages and store data in MongoDB.
* Implement an API for accessing and processing stored data.
* Provide an overview of different types of NoSQL databases, their benefits, and drawbacks in the report.
* Compare and evaluate the relational database approach with a key-value document store database (like MongoDB).
* Include MongoDB design, implementation, screenshots, installation, configuration, and data management system in the report.

## Problem Identification

IoThings, a startup business in the domain of home automation want a new NoSQL database that can deal with multiple devices and sensors. This database should be accompanied by a website that can be used to interact with the database, queries will provide the necessary functionality to interact with the database through website. Factors such as data security and providing feedback to users will be emphasized in the design of the database.

## Modelling and Designing Process

This section of the report discusses the planning and designing process concerned with the creation of the database and website being used for IoThings for their automated home solutions. The first element of the design process is to understand what type of devices are used in IoT homes, with this information it will provide greater clarification of what devices are relevant to the database.

Once this has been performed, different functionality for different devices will be planned and spread over a multitude of rooms. Going by the document-based NoSQL approach in MongoDB Collections. A website will then be created and connect to a database create any relevant queries. A flowchart of this process is presented in Figure 1.

A diagram of a flowchart

Description automatically generated with low confidence

Figure : Design Process Flowchart

The table (Figure 2: Table) below is concern with the modelling of the data used in the database. 14 different IoT devices are created and spread out amongst different rooms in a house.

|  |  |
| --- | --- |
| System (Device) | Attributes |
| Sound System | * Volume * Equalizer * Bluetooth connectivity * Wireless streaming * Multiple audio input options |
| Refrigerator | * Temperature control * Ice maker * Water dispenser * Adjustable shelves * Energy efficiency rating |
| Shower | * Water temperature * Water pressure * Shower head settings (spray patterns) |
| Garage Door | * Door open/closed * Remote control * Manual release mechanism * Energy Usage |
| Alarm | * Energy Usage * Alarm activation/deactivation * Security code settings * Entry/exit delay settings * volume |
| Television | * Energy Usage * Turn on/off * Brightness * Bluetooth Connectivity |
| Heater | * Energy Usage * Temperature * Ambient/Fan |
| Light System | * Energy Usage * Set lights on/off * Brightness |
| Ventilation Device | * Energy Usage * Ventilation on/off |
| Oven | * Energy Usage * Temperature * Cooking modes * Timer * Self-cleaning feature |
| Carbon Monoxide Detector | * Energy Usage * Alarm activation/deactivation * Battery level |
| Humidity Sensor | * Energy Usage * Alarm activation/deactivation * Battery level * HVAC Integration |
| Temperature Sensor | * Energy Usage * Alarm activation/deactivation * Battery level * High/low temperature alerts * Temperature trend tracking * Thermostat Integration |
| Motion Detector | * Energy Usage * Alarm activation/deactivation * Battery level * Motion sensitivity * Detection range * Pet-friendly Detection |

Figure : Devices Table of Attributes

## Relevant Software and Tools

For the creation of the database used to store this data, MongoDB will be used in tandem with the IDE (Integrated development environment) VSCode due to its simplicity and functionality. The addition of JavaScript will be present as well which will be responsible for the development of a few collection and insertions for the devices from the table stored in the document-based approach. The website will utilize MongoCompass in order to visualize any changes made to the database.

## Architecture of System

The API leverages the MongoDB Node.js driver to interact with the database and perform CRUD (Create, Read, Update, Delete) operations on device documents. MongoDB Compass, a visual database management tool, is used to interactively explore the data, query the database, and ensure data consistency and accuracy during development.

The API code includes a **connectToDb** function that establishes a connection to the MongoDB database. Upon successful connection, the API starts listening for incoming requests on port 3000. The **getDb** function returns the database connection object, allowing subsequent CRUD operations on the devices collection.

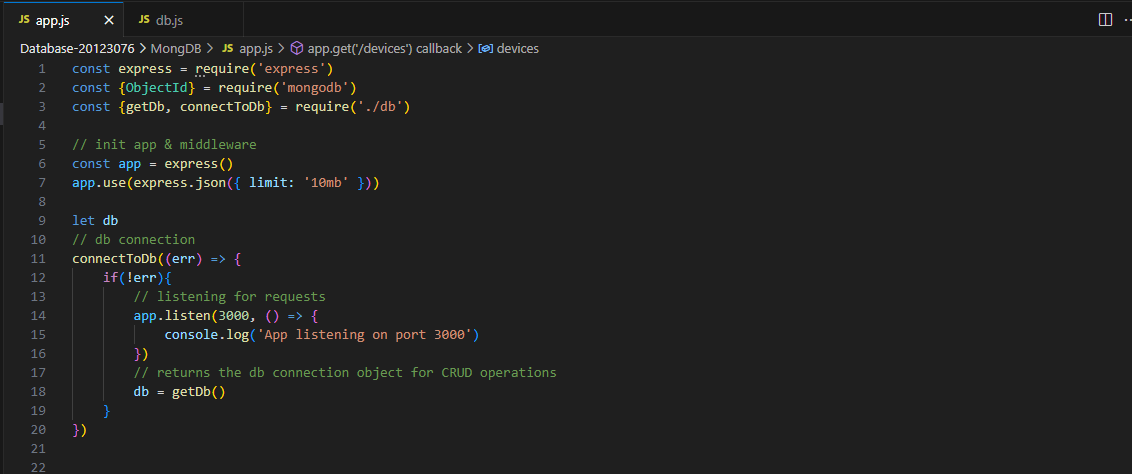


Figure : app.js part-1

The API defines several routes to handle different HTTP methods and perform corresponding database operations. These routes(that are found in the figures below) include:

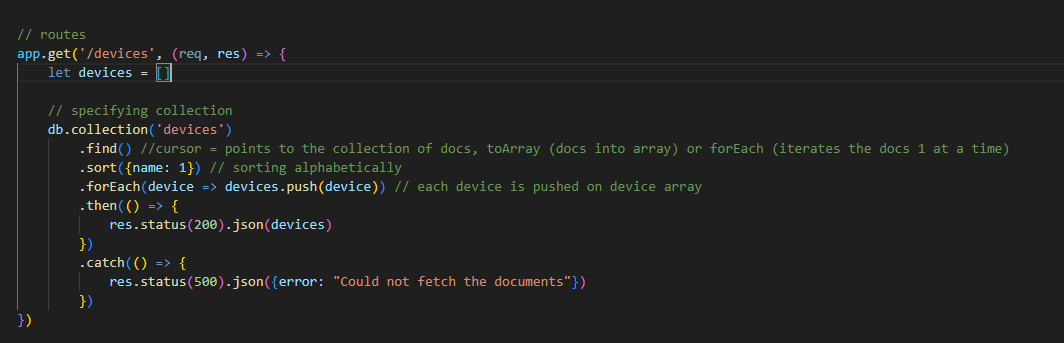


Figure : app.js part-2

**/devices:** This route handles the GET method to retrieve all devices from the database. The find method is used to fetch the documents, which are sorted alphabetically and returned as a JSON response.

**/devices/:id:** This route handles the GET method to retrieve a specific device based on its unique identifier (id). The findOne method is utilized to search for the device document with the corresponding \_id field and return it as a JSON response.

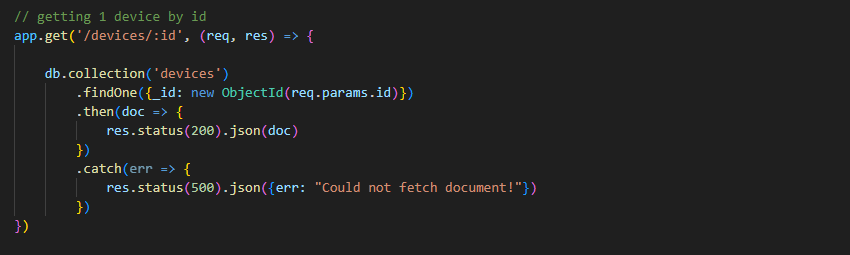


Figure : app.js part-3

A screen shot of a computer code

Description automatically generated with low confidence

Figure : app.js part-4

A screen shot of a computer program

Description automatically generated with low confidence

Figure : app.js part-5

**/devices:** This route handles the POST method to create a new device. The request body contains the device details, which are inserted into the devices collection using the insertOne method. The resulting document is returned as a JSON response.

**/devices/:id:** This route handles the DELETE method to remove a device based on its id. The deleteOne method is used to find and delete the document with the corresponding \_id field. The result of the deletion operation is returned as a JSON response.

**/devices/:id:** This route handles the PATCH method to update a device based on its id. The request body contains the updated device details, which are applied to the document using the $set operator in the updateOne method. The result of the update operation is returned as a JSON response.

A screen shot of a computer

Description automatically generated with medium confidence

Figure : db.js for mongodb connection

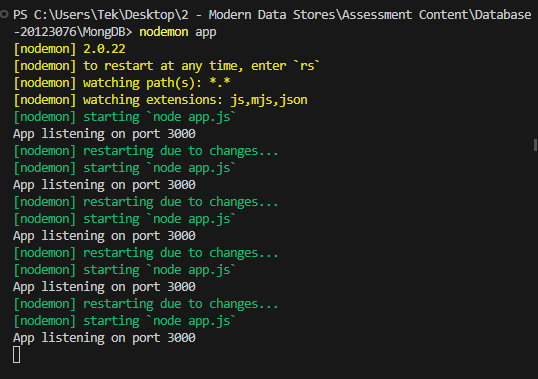


Figure : nodemon running on port 3000

## PostMan Utilization

Postman played a vital role in performing CRUD operations within the Device Management API, allowing for seamless testing and interaction with the API endpoints. The following are examples of how Postman was utilized for each CRUD operation:

Create (POST) Operation, With Postman, a new device could be easily created by sending a POST request to the /devices endpoint. The request body contained the necessary information for the new device, such as its name, type, and other relevant attributes. Postman allowed for easy customization of the request body, making it straightforward to test different data combinations and ensure the successful creation of devices.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : Creation Op via Postman

Read (GET) Operations, Postman facilitated retrieving a list of devices or accessing specific devices by their ID. By sending a GET request to the /devices endpoint, a list of all devices could be retrieved. Additionally, by specifying the device ID in the request URL, a GET request could retrieve the details of a specific device. Postman allowed for effortless parameterization of the request URL, enabling quick and convenient retrieval of device data.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : Reading Op via Postman-1

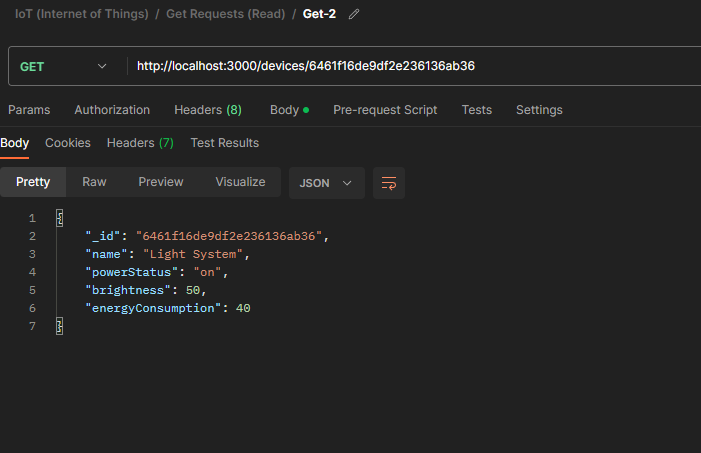


Figure : Reading Op via Postman-2

Update (PATCH) Operation, To update the variables of an existing device, Postman enabled the sending of PATCH requests to the /devices/:id endpoint. The request body included the updated attributes of the device. By specifying the device ID in the request URL, Postman ensured that the correct device was targeted for the update operation. This capability provided a straightforward way to test and validate the functionality of updating device information.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : Updating Op via Postman

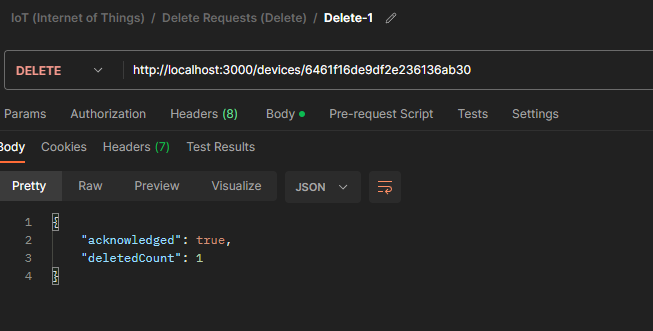


Figure : Deletion Op via Postman

Delete (DELETE) Operation, With Postman, device deletion was accomplished by sending a DELETE request to the /devices/:id endpoint, specifying the device ID in the request URL. Postman allowed for the easy configuration of headers and parameters, making it simple to send the DELETE request and verify the successful removal of the specified device.

### Conclusion

In conclusion, implementing a NoSQL database system for the marketing database of IoT companies offers significant advantages. NoSQL databases, such as MongoDB, provide flexibility in handling diverse and unstructured data generated by IoT devices. They offer high scalability, performance, and support for geographically distributed deployments, which are essential for IoT environments.

Investing in a NoSQL database system allows IoT companies to efficiently store and manage the vast amount of data generated by IoT devices. The flexible data model of NoSQL databases accommodates the dynamic nature of IoT data, eliminating the need for rigid schemas. Additionally, the scalability and performance capabilities of NoSQL databases ensure that IoT ecosystems can handle the growing volume of data and accommodate the increasing number of devices.

Looking ahead, building a customer front-end application to visualize stored sensor data is a valuable future consideration. This front-end application can provide users with an intuitive interface to explore and analyse the collected IoT data. Leveraging the querying and real-time analytics capabilities of NoSQL databases, users can gain valuable insights and make informed decisions based on the data. Integration of data visualization tools and dashboards can further enhance the usability and enable better utilization of the rich IoT data.

# References

*Vaughan-Nichols, Steven J. "MongoDB "open-source" Server Side Public License rejected". ZDNet. Archived from the original on January 16, 2019. Retrieved January 17, 2019.*

*"MongoDB's licensing changes led Red Hat to drop the database from the latest version of its server OS". GeekWire. January 16, 2019. Archived from the original on January 17, 2019. Retrieved January 17, 2019.*

*Krebs, Brian. "Extortionists Wipe Thousands of Databases, Victims Who Pay Up Get Stiffed". krebsonsecurity.com. Brian Krebs. Archived from the original on January 11, 2017. Retrieved January 11, 2017.*

*Constantin, Lucian (January 6, 2017). "Ransomware groups have deleted over 10,000 MongoDB databases". Computer World. IDG. Archived from the original on January 10, 2017. Retrieved January 11, 2017.*

*Ottenheimer, Davi. "How to Avoid a Malicious Attack That Ransoms Your Data". www.mongodb.com. Retrieved June 22, 2021.*

*"MongoDB Bind IP Compatibility". MongoDB. MongoDB. Archived from the original on March 6, 2019. Retrieved March 5, 2019.*

*Kyle Kingsbury (April 20, 2015). "Call me maybe: MongoDB stale reads". Archived from the original on August 15, 2015. Retrieved July 4, 2015.*

*"Release Notes for MongoDB 3.4". MongoDB Manual. Archived from the original on August 14, 2018. Retrieved April 6, 2018.*

*Kingsbury, Kyle (February 7, 2017). "MongoDB 3.4.0-rc3". Jepsen. Archived from the original on October 23, 2017.*

*"Atomicity, isolation & concurrency in MongoDB". scalegrid.io. Archived from the original on September 10, 2017. Retrieved June 28, 2017.*

*"MongoDB Goes Pluggable with Storage Engines". datanami.com. March 5, 2015. Archived from the original on July 4, 2017. Retrieved June 28, 2017.*