SenseMark – A database benchmark and evaluation study for alternative databases for Sensor data and IoT

Muhammad Arslan Mehfooz
Master’s Thesis

Autumn 2016
SenseMark – A database benchmark and evaluation study for alternative databases for Sensor data and IoT

Muhammad Arslan Mehfooz

December 26, 2016
Abstract

The emergence of cheaper sensors that can be connected in sensor networks and to the Internet has led to the concept of Internet of Things (IoT). By 2020 the estimate is that there will be tens of billions of data-spouting devices (sensors) that will be connected to the internet. This new technology trend will result in dramatic increase to Big data with high volumes and velocities. These data-spouting devices continuously capture, store, analyze and send data to cloud. These tiny devices are found everywhere today for example smoke detectors, smart cars, door locks, industrial robots, street lights, heart monitors, trains, wind turbines even in tennis racquets and toasters.

Traditional databases are typically not created to deal with the Big data aspects. Although database vendors tried to handle and store large volume of data with better performance but in the last decade, dramatically increase in size of data caused several issues to handle and store the data with faster access and less storage cost. NoSQL databases are considered to be an important component of Big Data when it comes to retrieving and storing large amount of data with reduced storage cast.

Companies and organizations tend to find a need to scale up quickly and efficiently to cover the dramatically increase in the data volume and demands of services they create. Most important factor to be considered is to find a cost-efficient solution that can meet the demands and requirements.

CITI-SENSE (project for air and environmental quality, 2012-2016) sensor data and ProaSense (project of oil and gas drilling) sensor data are considered as reference point to deal with and to store, evaluate and benchmark the performance using MongoDB (NoSQL), RDF (Graph Database), WFS and traditional SQL database solutions.

This report investigates the different database solutions available today. What do SQL, NoSQL and Graph databases offer? Comparison and differentiation? What are their pros and cons?

The purpose of this database benchmark and evaluation study is to figure out which database solution shows better performance for CRUD operations (Create, Read (Query), Update, Delete) with respect to response time, cost, storage capacity and scalability for sensor data.
Acknowledgements

First of all I would like to express my sincere gratitude to my supervisor Dr. Arne J. Berre at SINTEF for continuous guidance, encouragement, remarks, engagement and close monitoring during the whole thesis period.

This task is written at Research Institute SINTEF and I would like to thank the SINTEF- employees in the projects "CITI-SENSE" and "ProaSense" for all their help and guidance, particularly Nikolay Nikolov, Dina Sukhobok and Brian Elvesæter for spending time and sharing their knowledge and ideas with me.

I am also very grateful to Hårek Haugerud, Kyrre Begnum at Oslo and Akershus University College of Applied Sciences (HiOA) for spending time sharing their knowledge and ideas and would like to thank them and express how immensely important their lectures were.

Thanks to all of my classmates for all help, support and discussion throughout the study period. Also my sincere gratitude for University of Oslo (UiO) and Oslo and Akershus University College (HiOA) for providing high quality education.

Last, but not least, I would like to thank my family and friends for cooperation, moral support and encouragement.

Muhammad Arslan Mehfooz

December 26, 2016
## Content at a glance

1. Introduction ........................................................................................................... 2
   1.1 Problem Statement.......................................................................................... 3
   1.2 Thesis Structure ............................................................................................ 3
   1.3 Report Outline ............................................................................................... 4
   1.4 Motivation ...................................................................................................... 6
      1.4.1 Requirements.......................................................................................... 6
   1.5 Thesis Contribution ....................................................................................... 7
   1.6 Research Methodology ................................................................................. 7
   1.7 The work behind this report .......................................................................... 8
2 Background ............................................................................................................ 10
   2.1 Cloud computing ........................................................................................... 10
      2.1.1 Deployment models of cloud ................................................................. 11
      2.1.2 Infrastructure as a Service (IaaS) ......................................................... 12
      2.1.3 Cloud Service Provider today ............................................................... 12
      2.1.4 Amazon Web Services (AWS) .............................................................. 13
   2.2 The Internet of Things (IoT) ......................................................................... 14
   2.3 Big Data .......................................................................................................... 15
   2.4 NoSQL Databases and Big Data .................................................................. 16
      2.4.1 MongoDB (Document Store) ................................................................ 21
   2.5 RDF Linked Data/Triple Stores ..................................................................... 21
      2.5.1 Framework ............................................................................................. 22
3 Evaluation of existing Big Data Benchmarks ...................................................... 24
   3.1 State-Of-Art (Big Data Benchmarks) ............................................................. 24
   3.2 Analytic tools ................................................................................................. 29
4 Benchmark Requirements ..................................................................................... 30
   4.1 Proposed Solution ......................................................................................... 30
      4.1.1 SenseMark ............................................................................................. 31
5  CITI-SENSE case study ................................................................. 32
5.1  CITI-SENSE ........................................................................... 32
  5.1.1  Data Ingestion service ...................................................... 35
  5.1.2  Data publication/ Access Services ................................. 36
6  SenseMark – Concept and Design ........................................... 51
  6.1  Concept of SenseMark .......................................................... 51
    6.1.1  Target Scenario (Requirement) ...................................... 51
    6.1.2  Stressed Scenario .......................................................... 52
    6.1.3  Benchmark operations to be considered ....................... 52
6.2  Design of SenseMark ............................................................. 53
  6.2.1  Requirements for SenseMark Architecture ....................... 53
  6.2.2  Technical Challenges ....................................................... 54
  6.2.3  Proposed Approach ......................................................... 55
7  SenseMark – Implemented and Analyzed for MongoDB ............ 57
  7.1  Implementation ..................................................................... 57
  7.2  Benchmarking MongoDB .................................................... 58
    7.2.1  MongoDB Write Performance Benchmark .................... 58
    7.2.2  MongoDB Read Operations Performance Benchmark ...... 67
8  SenseMark – Considered for RDF/Linked Data ......................... 74
  8.1  DataGraft (RDF/Linked Data) .............................................. 74
  8.2  Performance Challenges with DataGraft / RDF ................... 77
9  Evaluation and Results ............................................................ 80
  9.1  Evaluation of SenseMark ...................................................... 80
  9.2  Results ................................................................................ 81
    9.2.1  Write Performance of MongoDB ................................... 82
  9.3  Read Performance of MongoDB as compared to WFS ........... 82
  9.4  RDF/Linked Data ............................................................... 83
10  Conclusion and Future Work ................................................... 85
   10.1  Conclusion ........................................................................ 85
   10.2  Future Work .................................................................... 86
11  Appendix ................................................................................. 88
12  Bibliography ........................................................................... 94
List of Figures

Figure 2.1 Different Cloud Deployment Models[8] ......................................................... 11
Figure 2.2: Decline in dominance of SQL ................................................................. 18
Figure 2.3: Comparison of ACID and BASE ............................................................... 19
Figure 2.4: CAP theorem principle[24] ...................................................................... 20
Figure 2.5: NoSQL Database - Storage Architecture[25] ............................................. 20
Figure 2.6: The screenshot of Grafterizer’s GUI ......................................................... 22
Figure 2.7: RDF mapping in Grafterizer ................................................................... 23
Figure 5.1: CITI-SENSE Outdoor Data Portal ......................................................... 33
Figure 5.2 Citizen Observatory Toolbox[47] ............................................................. 34
Figure 5.3: Architecture of CITI-SENSE platform .................................................. 35
Figure 5.4: CITI-SENSE Data Model Schema for WFS ............................................ 37
Figure 5.5: All observations of sensor-id AT_5 ......................................................... 38
Figure 5.6: All observations of sensor-id AQ_687150 .............................................. 39
Figure 5.7: All observations for sensor-id AT_11 at a specific point in time .......... 40
Figure 5.8: Observations of Sensor-id AT_11 between specified time period .... 41
Figure 5.9: All Observations of sensor-id AT_10 between specified time period .... 41
Figure 5.10: Last Observation of sensor-id AT_10 .................................................... 42
Figure 5.11: WFS service for OSLO 24 hours’ data .............................................. 47
Figure 5.12: Oslo 24 hours ..................................................................................... 47
Figure 5.13: Time out error while calling Oslo 24 hours’ measurements .......... 47
Figure 5.14: Oslo one hour WFS service facing time out issue .............................. 48
Figure 5.15: Oslo one hour WFS Service ................................................................ 48
Figure 6.1: Conceptual Architecture of SenseMark ............................................... 53
Figure 6.2: Components of Sensing Architecture ................................................... 55
Figure 7.1: Average write events of 2000 sensors sending 2000 events/s .............. 59
Figure 7.2: 2000 write events/sec performed 3 times ............................................. 60
Figure 7.3: Average write events of 5000 sensors sending 5000 events/s .......... 61
Figure 7.4: 5000 write events/sec performed 3 times ............................................. 62
Figure 7.5: Average write events of 10000 sensors sending 10000 events/s .................... 63
Figure 7.6: 10000 write events/sec performed 3 times .................................................. 63
Figure 7.7: Average write events of 20000 sensors sending 20000 events/s .................. 64
Figure 7.8: 20000 write events/sec performed 3 times .................................................. 65
Figure 7.9: Average write events of 50000 sensors sending 50000 events/s ............... 66
Figure 7.10: All Observations of a specific sensorId ....................................................... 67
Figure 7.11: All observations in a specific time period ................................................. 70
Figure 7.12: All observations for a specific sensor-id in a specific time period .......... 71
Figure 7.13: Last observation for a specific sensor-id ..................................................... 72
Figure 7.14: Last observation for a specific sensor-id for a specific observed property ..................................................................................................................... 72
Figure 8.1: Grafterizer GUI in DataGraft ................................................................. 74
Figure 8.2: RDF Mapping of CITI-SENSE CSV ......................................................... 75
Figure 8.3: Overview of Clojure Code ........................................................................ 75
Figure 8.4: Downloadable executable in DataGraft ................................................... 76
Figure 8.5: Sample of RDF file ................................................................................. 76
Figure 8.6: SPARQL Query ....................................................................................... 77
List of Tables

Table 1.1: Abbreviations and Acronyms ............................................................... x
Table 2.1: Classification of NoSQL databases .......................................................... 21
Table 3.1: State of art (Big Data Benchmarks) ........................................................ 28
Table 5.1: CITT-SENSE Indoor Data Portal .............................................................. 33
Table 5.3: Results of all observations of a specific sensor-id ...................................... 39
Table 6.1: Non-functional requirements from sensing and storage layer ..................... 54
Table 7.1: MongoDB write performance test for CITT-SENSE real scenario .............. 59
Table 7.2: MongoDB write performance test criteria for 5000 sensors ...................... 61
Table 7.3: MongoDB write performance test criteria for 10000 sensors ..................... 62
Table 7.4: Total no. of 9 days’ data events in MongoDB .......................................... 70
Table 9.1: Comparison of SenseMark with an ideal Big data benchmark ................. 80
Table 9.2: Evaluating SenseMark for CITT-SENSE requirements .......................... 81
Table 9.3: Write performance results of MongoDB .................................................. 82
Table 9.4: Read performance results of MongoDB ................................................... 83
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIF</td>
<td>European Interoperability Framework</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>EDA</td>
<td>Event Driven Architecture</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IPR</td>
<td>Intellectual Property Rights</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>OPC</td>
<td>OLE for Process Control</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>REST</td>
<td>Representational state transfer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Drive</td>
</tr>
<tr>
<td>SSN</td>
<td>Semantic Sensor Network</td>
</tr>
<tr>
<td>WFS</td>
<td>Web Feature Service</td>
</tr>
<tr>
<td>GEOSS</td>
<td>Global Earth Observation System of Systems</td>
</tr>
<tr>
<td>GML</td>
<td>Geography Markup Language</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>SEDS</td>
<td>Spatial Environmental Data Server</td>
</tr>
<tr>
<td>INSPIRE</td>
<td>Infrastructure for Spatial Information in the European Community</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>LOD</td>
<td>Linked Open Data</td>
</tr>
<tr>
<td>O&amp;M</td>
<td>Observations and Measurements</td>
</tr>
<tr>
<td>OGC</td>
<td>Open Geospatial Consortium</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SDI</td>
<td>Spatial Data Infrastructure</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol And RDF Query Language</td>
</tr>
<tr>
<td>SSNO</td>
<td>Semantic Sensor Networks Ontology</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>VGI</td>
<td>Volunteered Geographic Information</td>
</tr>
</tbody>
</table>

Table 1.1: Abbreviations and Acronyms
I

INTRODUCTION
1. Introduction

The evolution of Internet of Things (IoT) implies a need for storage and processing of large amounts of sensor data that typically has spatiotemporal characteristics. In this context, there are several multiple database solutions that are being proposed (including SQL alternatives such as No-SQL and RDF/Graph databases etc., and combinations with distributed cloud solutions and parallel processing i.e. Hadoop etc.), but it is vague from a functionality and performance point of view about their advantages and disadvantages.

Relevant problem examples from this area can be found in the CITI-SENSE[1] project with air quality sensor data, and the ProaSense[2] project with Oil Drilling and Manufacturing equipment sensor data. With these as representative example problems, the objective is to create a database benchmark for representative sensor data and to benchmark alternative database solutions with respect to the functionality, query and performance characteristics. Alternative databases solutions include SQL databases (such as MySQL) as a reference point, SpatioTemporal Feature databases (such as the OGC WFS and SOS standard), NoSQL document databases (such as MongoDB and CouchDB), and triple stores for RDF/Linked Data (such as DataGraft/Ontotext).

The work will be to first establish a relevant and representative data model for sensor and IoT data and a related set of relevant operations for storage, queries and processing of data from this model, typical for the requirements of the CITI-SENSE and ProaSense projects. Relevant candidates as input for this IoT / sensor model is the OGC SensorML, OGC/ISO Observations and Measurements, IETF SenML (used in the SINTEF SensApp MongoDB solution), W3C SSN and others. Examples of relevant operations can be found through the applications identified in the CITI-SENSE and ProaSense projects.

This project will include reviewing of existing database benchmarks to identify if some existing database benchmarks can be used directly, or create/adapt a new database benchmark for the identified needs.

Based on representative data according to the selected data model – perform database benchmark evaluations for selected databases from the various database groups, SQL, NoSQL, WFS/SOS, RDF/Linked Data, this project will provide a comparative description of these alternative databases related to their functional differences and performance characteristics for different kinds of operations and queries.
In the end, this project will provide a recommendation for future projects that what databases are adequate for different situations.

1.1 Problem Statement

The following problem statement will serve as guideline to materialize the vision of this thesis:

“To evaluate and benchmark different database solutions for representative sensor data for optimized performance with respect to the classical CRUD operations (Create, Read (Query), Update, Delete) with respect to response time, cost, storage capacity and scalability”.

The term optimized is used widely in information technology and is used to explain the process of attaining better performance, to make something run faster without delay and error, and to take advantage of the resources available e.g. time, cost, scalability. It is observed that several of these requirements are compromised majority of the times.

To answer this problem statement, several assumptions have been made based on different database solutions and available benchmark tools. The answer to this problem is stated in Design and Approach section in which a benchmark tool SenseMark is designed and implemented on different database solutions to analyze the performance of each for classical CRUD operations (Create, Read, Update, Delete).

1.2 Thesis Structure

The thesis is divided into two parts, each part is described in the following section.

Part I

This part consists of first 5 chapters and describes the method of work followed by an example for the problem definition, requirement to solutions and finally evaluation of existing database solution and Big Data benchmarks.

CITI-SENSE architecture and platform is considered as a reference point to access data ingestion services and data publication services for such type of sensor data events. In this part, currently available data ingestion and data publication services for such type of sensor data events are evaluated and their performance is analyzed for classical CRUD operations.
Data ingestion services which are deployed on Amazon AWS and contain single WFS end point that could be used by data providers to register sensors and observations. Data publication services consist of two type of web interfaces realized on Amazon AWS that allow users to access sensor data stored in the CITI-SENSE platform which is based on the following:

- WFS (Web Feature Service)
- REST (RESTful service)

Part II

This part consists of chapters 6-10. Based on the needs identified after using and evaluating currently available Big data benchmarks, a new database benchmark called SenseMark is designed/created and implemented on different database solutions to measure the performance for classical CRUD operations with regards to volume (i.e. scaling of sensor data), velocity (i.e. speed of moving data and optimized reaction time) and variety (i.e. supporting heterogeneous type of data) of such type of sensor data.

Database solutions considered are as follows:

- SQL
- MongoDB (NoSQL)
- RDF/Triple (Graph Database)

1.3 Report Outline

This section shortly describes the structure of report and provides an overview of the parts of the research that are discussed in each chapter.

- **Chapter 1 – Introduction** introduces the need to evaluate and benchmark different storage solutions for sensor data and to suggest a best storage solution for such type of sensor data events (IoT) for classical CRUD operations.

- **Chapter 2 – Background** describes related work and background knowledge about Big Data and IoT. Properties of different database solutions including SQL and NoSQL are discussed and highlighted in this part, their pros and cons and to figure out which one could be best for sensory Big data. Linked data RDF/Triple is also discussed in this part.
• **Chapter 3 – Evaluation of existing Big Data Benchmarks** explore currently available and existing Big Data benchmarks, discuss the properties of each of them and compare with each other. This chapter also provide description of analytic tools that are used in this project.

• **Chapter 4 – Requirements to solution** highlights the need to design and create a new benchmark tool *SenseMark* after evaluating the currently available Big Data benchmarks.

• **Chapter 5 – Relevant Research** consists of exploring CITI-SENSE platform. Evaluation of both data ingestion and Data publication services offered in CITI-SENSE architecture. Also highlight the performance challenges faced in CITI-SENSE WFS service.

• **Chapter 6 – SenseMark - Concept and Design** display the design of SenseMark, it also highlights the need and requirement from SenseMark architecture.

• **Chapter 7 – SenseMark Implementation for MongoDB** describes the implementation of SenseMark on MongoDB and performance results with respect to classical CRUD operations.

• **Chapter 8 – SenseMark considered for RDF/Linked Data** explore the implementation and evaluation of RDF/Linked Data. Results shown by RDF and DataGraft for such type of sensor data and IoT.

• **Chapter 9 – Evaluation and Results** contains the evaluation of MongoDB, RDF and WFS. Discuss the advantages and disadvantages of each storage solution and discuss the results of MongoDB as compared to WFS and RDF/Linked Data.

• **Chapter 10 – Conclusion and Future work** summarizes this thesis, provides an overview of the contributions in terms of expected results and acquired results, at the end suggest the direction for future work.

• **Acronyms**: The acronyms used throughout this paper are mentioned in this chapter for convenience to the reader. The acronym chapter is placed after the Table of Contents.
1.4 Motivation

Motivation and goal behind this project is to provide a sensing architecture that is capable of supporting large amount of sensor data events and to suggest the best and appropriate database solution for such type of sensor data (IoT), to ensure future scalability, flexibility, reliability, availability and maintainability with regards to volume, velocity and variety and to cope with huge amount of different sensor data events (big data) in real-time.

The goal behind this evaluation and research work is:

1. to analyze and compare the results shown by different databases
2. to discuss the advantages and disadvantages of each solution.
3. to benchmark each of them to find out that in which circumstances and conditions each of them is more appropriate to use.

Following are the specific characteristics that are required for the database system to be acceptable for such type of sensor data (IoT).

1.4.1 Requirements

Requirements fulfilled in the scope of this thesis are as follows:

- It should be possible to take backups and restores without any down time.
- The system must handle only minor failures, for example, a single server crash in a cluster without effecting the performance of whole database system with zero downtime.
- The system must be scalable enough to manage large and rapid increase volume of data for such type of sensor data (IoT) that continuously send to server making the database stressed that is the actual big challenge.
- The system is desired to be cost efficient and must be able to scale elastically so that new servers could be added to the system to ensure scalability without downtime and less storage cost.
- The system must be efficient to provide the requested results (Query operations) accurately and quickly.
1.5 Thesis Contribution

The thesis contributes by providing a foundation for a Big data benchmark framework for sensor data called SensMark, and through the demonstration of the usage of this for the NoSQL database MongoDB. The thesis further contributes to the evaluation of DataGraft [3] and Grafterizer for sensor data – a web-based tool and framework for data transformation and cleaning and mapping into triple/RDF. DataGraft is a powerful cloud-based platform for data transformation, publication and hosting of data.

The thesis contributes evaluation of WFS service (CITI-SENSE) MongoDB database solution (NoSQL) and RDF/triple store (DataGraft) by designing a benchmark SenseMark to evaluate the performance of each storage solution for sensor data.

Summary of Thesis contribution

Summary of thesis contribution include:

- Evaluating CITI-SENSE architecture and framework consisting of a WFS storage solution to figure out the performance challenges faced in CITI-SENSE.

- Evaluating the graph database solution using DataGraft for such type of sensor data (IoT) and implementing required data cleaning and transformation operations and to map CITI-SENSE data into RDF/triples and publishing it on DataGraft server.

- Development of SenseMark – a benchmark tool for generating sensor data and storing it to database solution.

- Implementation of SenseMark on the NoSQL document database MongoDB to generate and store sensor data according to CITI-SENSE requirements and benchmarking performance for classical CRUD operations.

1.6 Research Methodology

The research method for development and evaluation of MVAP is based on the approach for technology research [4]. Three steps are defined in the method for technology research, to either improve existing artifacts or make new ones. All three steps are implemented during this thesis and consist of:
1.7 The work behind this report

The database management systems examined in this project were not well known to me when I started working on them. Especially for RDF/Linked Data, the data model was quite unknown to me. Plenty of time was spent in understanding the working principle of Linked data, how to operate and use the Graph database management system and tool (DataGraft) used to map the data into triples(RDF). Also, spent a lot of time in doing jar transformation of CSV files and publishing them to DataGraft server due to scalability issues.

To understand and get better knowledge about the relational database management system, I joined a project group CITI-SENSE in which WFS on the top of a relational database system is used to host sensor data on a SQL based server managed by a company SNOWFLAKE. I participated in the discussions and telephonic conferences where the issues in the current database systems were highlighted and discussed in detail. I got good understanding and knowledge about the problems and issues that occur while using relational database management systems for sensor data especially in CITI-SENSE case.

As a starting point, the already available solution (WFS) for CITI-SENSE data was evaluated and benchmarked for five relevant scenarios that users want to get result for. Those Scenarios are listed below.

- Scenario 0: Publish/Write the observations for a specific sensor device.
- Scenario 1: Query all the observations for a specific sensor device.
- Scenario 2: Query all observations for a specific sensor device at a specific point in time (“snapshot”).
- Scenario 3: Query all observations for a specific sensor device in a given time period.
- Scenario 4: Query the latest observation for a specific sensor device.
- Scenario 5: Query the latest observation for a specific sensor device for a specific pollutant.

The aforementioned scenarios are the CITI-SENSE user requirements that in the project were facing various performance challenges. While calling each of these services, users of CITI-SENSE platform were unable to get accurate query results due to some performance challenges in the WFS service. The proposed solution to overcome this problem was to try an alternative database solution and for that purpose, MongoDB and RDF/Linked data were considered as good alternative
database solutions.

During investigation and evaluation of relational database model and MongoDB model, I found that existing benchmarks didn’t cover requirement and needs for sensor data and NoSQL databases like MongoDB so I felt that there is a need to create/design a new benchmark tool SenseMark to evaluate the performance of alternative database solutions such as MongoDB for sensor data which led me to a new evaluation process.
2 Background

This chapter provides an overview and background knowledge about Big data, sensor devices (IoT) and the relationship between them. Concept of cloud computing and different cloud service providers is also discussed in this part. This chapter shows an overview of different database solution, their pros and cons in order to figure out that which one could be used as an alternative database solution for CITI-SENSE. NoSQL databases are highlighted and discussed as they are considered to be more efficient and effective in such cases where data volumes are continuously increasing. Furthermore, this chapter also covers working principle of graph database solution RDF/Linked data using DataGraft.

2.1 Cloud computing

Cloud computing is an evolving paradigm that is being transformed and delivered in a manner similar to traditional utilities for example electricity, gas and telephone. In this way, user access the service without concerning where it is hosted and how it is delivered [5]. Cloud computing gained traction in 2007. and since then the research study has been conducted substantially. In The National Institute of Standards and Technology defines the term "cloud computing" as follows[6]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The most common form of Cloud computing is Infrastructure as a Service (IaaS) where it is possible to provision things like virtually hosted machines and storage space on which the whole system is built. There could also be higher levels, Platform as a Service (PaaS) or Software as a service (SaaS) where things like database and even games are provisioned. Thus, Cloud is an alternative to purchase physical hardware systems and storing them in data center.

The five most important characteristics of cloud computing are defined in the The NIST Definition of Cloud Computing [7], as follows:

**On-demand self service**

The consumers can provision the resources and the computing capabilities they need such as server time and network storage. Consumer does not need to interact
with service provider but can provision automatically when they need.

**Broad network access.**

Capabilities are available over the network through standard mechanisms that promote use by platforms e.g. smart phones, tablets, laptops and workstations.

**Resource Pooling**

The provider uses a multi-tenant model where the hardware is shared through virtualization. The process is transparent to the end user who has no control over locality of resources.

**Rapid elasticity**

Resources can be provisioned and released rapidly and elastically, sometimes even automatically to scale rapidly outward and inward with demand.

**Measured service**

The utilization of resources can be monitored, controlled and reported, providing transparency to both the end user and the service provider.

2.1.1 Deployment models of cloud

Deployment models of cloud infrastructure are of following four types [7]:

- **Private Cloud** Infrastructure is used by a single organization and is not accessible to outside users. **Community Cloud** is provisioned for exclusive use by only one specific community. **Public Cloud** is open to everyone and could be used by general public. This type of cloud infrastructure is operated by businesses, academics and government organizations. Another type of cloud infrastructure is **Hybrid Cloud** which is the composition of two or more distinct cloud infrastructures (Private, community or public)[8]

![Figure 2.1 Different Cloud Deployment Models](image-url)
2.1.2 **Infrastructure as a Service (IaaS)**

Infrastructure has shifted from hardware to software and it refers to infrastructure as a service (IaaS) which provides all the processing power, storage, memory. All these sources combine together to form virtual machines (VMs), routers, and other components that are usually part of physical systems. This is commonly called cloud.

The main reason why businesses may want to move to cloud infrastructure is that it is easier to manage, to get full control over environment, cost effective, resolve the scalability issue and saves time. Cloud servers may use to run only when required and terminated or stopped to cut the cost. Thus, in this way it requires less monitoring and provisioning when the system is not in use or less used. Another advantage is to setup auto scaling and alarms so that when system get more traffic than its maximum capacity, it scales up automatically to fulfill the requirements[9]. In the same way datacenters, have also moved to cloud to host the data and don’t need to worry about the scalability and need to pay for the amount of hosted data.

2.1.3 **Cloud Service Provider today**

There are different cloud service providers providing different services. The most popular IaaS cloud providers are Amazon with Amazon Web Services, Microsoft with Azure, Google with compute Engine, VMWare with vCloud are a few of them that are most popular in the market today. Choosing the cloud service provider depends upon the customer needs and requirement as different vendors are specialist and famous in different cloud services.

Amazon Web Services is the most popular in IaaS market due to the highest share, highest compute capacity in use as compared to other Infrastructure as Service (IaaS) providers.

Microsoft Azure is recommended and more suitable for general business applications and development environments for Microsoft-centric organizations and also used in cloud-native applications and batch computing. Google cloud is also recommended for cloud-native applications and batch computing as well as projects that leverage Google cloud platform as a whole.[10].
2.1.4 Amazon Web Services (AWS)

Amazon Web Services (AWS) is a cloud computing platform that consists of many remote computing services available to consumers. With AWS, one can find a complete set of highly available services that are designed to work together to build sophisticated scalable systems and applications. AWS is a global service that is distributed across different regions in North America, Brazil, Asia, Australia and Europe and providing web-based storage and computing services to developers [11].

Storage on the virtual servers is ephemeral storage which means that if the instance is terminated or shutdown, the data is wiped forever. In Amazon, it is possible to provision a hard drive on Storage Area Network and connect them to EC2 instance that is known as Elastic Block store (EBS).

Amazon services are providing best practices to their users that are as follows:

- Multiple availability zones to ensure high availability
- Replication between regions to improve continuity
- Geographical expansion covering whole World.
- Providing highly automated and monitored infrastructure

There are a lot of services offered by AWS in different categories, e.g. compute, storage & content delivery, database, networking, developer tools, management tools, security and identity, analytics, internet of things, game development, mobile services, application services and enterprise applications. All of these categories combine to provide a fully automated, monitored, highly available and functional infrastructure to organizations [12].

Amazon S3 (Storage)

There is another storage service offered by AWS which is called Simple Storage Service (S3). S3 provides highly scalable, safe, secure object storage to IT teams and developers. By using this service, one is allowed to upload data and pay for the storage resource is used. S3 has additional features like static website hosting [11].

CITI-SENSE has all of its sensor data stored in Amazon S3 (Simple Storage Service) in EU-West-1 which is the only European region and is located in Ireland. The EU-West-1 region consists of three availability zones (AZ). These are physical server centers that are located at minor network latency of one millisecond.

Amazon S3 provides a simple web-service interface that can be used to
store, retrieve any amount of data regardless of the time and location by using web anywhere. S3 has the capability to be used alone or along with Amazon EC2/ EBS, Amazon Glacier or third party storage repositories to provide a cost-effective object storage different use cases, such as cloud applications, content distribution, backup and restores, and big data analytics[13].

Amazon S3 is considered to be the ideal big data object storage. S3 is highly available big data hosting solution because it can scale automatically in order to meet customer demand, for example it can add any amount of content that can be accessed from anywhere from the web. Some key features of S3 include Security and Access Management, Lifecycle Management, Cost Monitoring Controls, Transfer of data to and from S3 with ease and REST interface in which standard HTTP requests are used to create, fetch, and delete buckets and objects[14].

### 2.2 The Internet of Things (IoT)

The Internet of Things value chain covers all those devices that are intelligent and connected. The IoT is the term that refer to increasing trend of using sensors in devices and objects to make them capable of communication and sending or receiving information[15]. The reasons could be various for increased used of sensors in devices by organizations. Sensors are used from small devices e.g. smart phones to bigger one like vehicles etc. Sensor may vary in type, shape and size. In a motor car, there are various sensors of different types, for example a temperature sensor in engine generate alarm if the temperature of engine crosses the set threshold to notify the driver that something is wrong.

There are other industries as well who get benefit from sensors. Some sensors are capable of sending data to manufacturers to inform them about the fault or issues in their equipment so that it could be resolved. In future, there will be huge increase in usage of sensors. Cisco predicted and released a report in 2011 that there would be 25 billion devices on internet by 2015 which will increase to double (50 billion) by 2020[16].

Another leader in this race, Siemens, has said that these smart things are starting to power a fourth Industrial Revolution (after steam, electricity and wired computers) [17]

Today, Big Blue that is the nick name for IBM, is putting that tiny technology to work, developing a multi-application gas sensor that could help airports to detect and track biochemical threats, determine whether the steak in your
fridge has spoiled, or even diagnose breast cancer and other diseases simply by analyzing your breath.

Technology landscape is continuing to grow rapidly. Facebook touches about 500 million users, mobile phone users have reached to 4 billion, and the number of internet users in the mobile environment have also reached to 450 million. In the same way, information technology has also changed its way of deployment like increased use in cloud computing and virtualization. This rapid shifting of technology environment has also promoted a term Internet of Things (IoT) that has the ability to capture, compute, communicate and collaborate. These IoT devices are embedded with sensors, actuators and communication capabilities and will soon be able to send a massive and huge amount of data on large scale[18].

With the emergence of standard of safety, some core technologies for the IoT are becoming more widely used. Automobiles insurance companies in Europe and United States are testing and installing sensors in the customer’s vehicles that would be able to sense the driving behavior of the driver to charge them according to it. Also some Luxury automobile manufacturers are using advanced sensors to react automatically in a situation when accidents are about to occur.[18]

2.3 Big Data

Social networks, web analytics, intelligent e-commerce usually need to manage data at a scale too big for traditional Relational Database Management Systems (RDBMS). Although database vendors tried to handle and store large volume of data with better performance but in the last decade, dramatic increase in size of data caused several issues to handle and store the data with faster access and less storage cost. NoSQL databases are considered to be an important component of Big Data when it comes to retrieving and storing large amount of data with reduced storage cost [19]. For example, according to report, traditional RDBMS SAN storage costs average $30,000+ per terabyte, whereas storage for NoSQL databases average $1000 per terabyte.

The Oxford English Dictionary defines big data as follows [20]:

*Extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions.*

Big Data is playing a vital role to improve modern society. People can make use of new technologies and tools to improve every aspect of life including, health, medical care, to accelerate discovery and innovation. Besides this improvement and positive aspects, Big Data is presenting various challenges to both Governments
and Citizens because these data technologies are becoming very pervasive and
difficult to understand. Also, there is danger of misuse and abuse of these large
datasets. So, there are various issues also faced due to this dramatic increase in data
sets giving rise to Big data. It is very important to take initiatives to secure these
data sets and sensitive information and also to make sure that database technologies
are used effectively and responsibly[21].

The emergence of wireless sensor networks has helped to give a quick rise
to the amount of data stored. These wireless sensor networks are continuously
transmitting data and has given birth to term big data which is widely recognized
trend now. This term Big Data is not only concerned with data volume but also
with the high-speed transmission and the various different information that are
difficult to collect, store, read update and delete using different available storage
solutions. All the data collected and generated by individual sensors is not
considered to be important and this type of data generated and captured by various
sensors is capable of producing large amount of data volumes that could be difficult
to handle for classical CRUD operations (Create, Read, Update and Delete) in an
efficient manner [12].

Data volumes are flooding at a very high rate and doubling every 18 months
[18]. Different tools and technologies are available to capture and analyze
information but organizations are trying to take the data use to new levels to attain
more better performance and availability in a cost-efficient way which is becoming
a big challenging It is also obvious that this dramatic data increase rate will
accelerate more in upcoming months due to increased usage of Internet of Things
continuously sending data to make the data volume flooded every second.

2.4 NoSQL Databases and Big Data

Due to development of Internet and cloud computing, term Internet of Things
(IoT) is used widely now a days and it has given dramatic increase to data volumes,
therefore it is required to have database solutions capable of storing and processing
such Big Data more efficiently and effectively[22]. Database solutions are required
to provide high performance for classical CRUD operations for which traditional
databases are facing challenges when it comes to store and query dynamic user
data. In this case NoSQL databases claim to be more efficient and faster as
compared to RDBMS in various use cases and especially when we discuss about
big data stores where data increases rapidly and continuously. Due this emergence
of new applications and technologies, databases are required to cover the following
demands [22]
- High concurrent of reading and writing with low latency
- Efficient big data storage and access requirements
- High scalability and high availability
- Lower management and operational costs

Although relational databases have provided solution to store data and vendors tried to update database systems to cope up with the latest demands but still traditional database systems failed to provide storage solution efficiently and effectively according to user requirements due to the following reasons[22]:

- Slow reading and writing
- Limited capacity
- Expansion difficult

NoSQL stands for Not Only SQL and this term is used for databases that are alternative to RDBMS (e.g. Oracle, MS SQL Server and IBM etc.). NoSQL databases are capable of storing larger amount of data. NoSQL databases are distributed, source of quick retrieval of information and portable as well. Another important feature of NoSQL databases is open source therefore, its code is available to everyone and can be modified, and complied according to requirements. NoSQL databases show high performance in linear way and are horizontally scalable. NoSQL doesn’t organize its data in related tables [23].

Although traditional Relational Database Management Systems (RDBMS) are widely used for decades and database vendors constantly tried to improve them to handle and support large volume of data. However new category of database technology called NoSQL is capable of supporting, handling of larger volumes of data with faster access and less storage cost. NoSQL databases are considered to be an important component of Big Data when it comes to retrieving and storing large amount of data. Another example of use case of NoSQL is to store huge amount of data in Facebook (which keep on increasing every second). NoSQL is becoming more and more popular due to its high storage capacity and is widely used now a day e.g.

- Google (BigTable, LevelDB)
- Linkedin (Voldemort)
- Facebook (Cassandra)
- Twitter (Hadoop/Hbase, FlockDB, Cassandra)
- Netflix (SimpleDB, Hadoop, HBase, Cassandra)
- CERN (CouchDB)
NoSQL databases use BASE theorem for data consistency whereas RDBMS use ACID theorem. Another advantage of NoSQL databases over RDBMS is that NoSQL can scale both horizontally and vertically while RDBMS can scale only vertically.[19]

Relational databases show mechanism to run and operate on a single machine, thus a single powerful and large machine is required to scale. In this case as the whole database is dependent on single machine and if the machine goes down, the whole database system goes down. A solution to this problem is to buy several small machines instead of single machine and create a cluster consisting of several machines to store data. This process is considered to be cheaper and horizontal scalable. In this case if one machine goes down, other machines maintain the reliability of cluster quite high. This is the mechanism shown by NoSQL database and that is one reason that NoSQL databases are become quite popular now a days [23].

![Figure 2.2: Decline in dominance of SQL](image)

Some important features followed by NoSQL databases are as follows.

**A. Acid free**

Better performance and scalability in NoSQL is achieved by sacrificing ACID compatibility. ACID stands for Atomicity, Consistency, Isolation and Durability. Basically, ACID concept comes from SQL environment but due to consistency factor, NoSQL solutions avoid to use ACID concept. AS NoSQL databases are based on distributed systems and data is spread to various machines in the cluster and it is required to maintain consistency. For example, if there is a change in one table, it is required to make changes in all machines on which data resides. Consistency could be attained if the information about the update process spread immediately through the whole system otherwise inconsistency is carried out and in this way ACID concept create trouble to NoSQL solutions.
B. BASE

BASE is reverse of ACID and this term stands for Basically, Available, Soft state and eventual consistency. Use replication and sharding to reduce the likelihood of data unavailability and use sharding. As a result, the system is always available even if subnets of the data are down and unavailable for short period of time. Thus, in general availability of BASE is achieved through supporting partial failure without whole system failure. As an example, if we discuss about the bank databases in banks and two people try to access same bank account from two different locations (Cities) then it is not only required to update data in time but require some real-time databases as well. Some other examples of same situation could be online ticket booking, and online shopping platforms.

![Figure 2.3: Comparison of ACID and BASE](image)

C. CAP

CAP stands for Consistency, Availability and partition Tolerance. CAP theorem is based on these three principles.

- Data should be consistent and available on all machines should be same in all respect and update process should run on all machine frequently.
- Data should be highly available to clients and must be accessible any time.
- Due to system failure and fault in nodes, database must work fine despite physical network partitions i.e. partition tolerance.
There are different types of NoSQL databases and each has its own set of features and characteristics, and these leads to the performance difference.

Different breeds of NoSQL databases:

- Key-values Stores
- Column Family Stores
- Document store Databases
- Graph Databases

NoSQL databases provide performance gains but some researchers are skeptical about data consistency.
2.4.1 **MongoDB (Document Store)**

MongoDB is a popular NoSQL type of database and is open-source. Main properties of MongoDB include document-oriented storage layer, auto-sharding behavior and asynchronous data replication between indexing in the form of B-trees and servers [26]. MongoDB is relatively new breed in database solutions and it shows no concept of tables, schemas, or rows. It doesn’t have features of traditional database management systems that we used in past e.g. foreign keys, joins, ACID compliance and transactions. MongoDB is the database that showed largest increase in the execution time due to locking mechanism but the reads are not exclusive. So, the mapping of records in memory increases performance. MongoDB is considered to be more efficient and best solution for sensor data due to its high performance, high scalability, high flexibility and low complexity.

MongoDB offers a flexible database schema due to which property documents in the same collection can be made up of different structures. Another most useful and important feature offered by MongoDB is its sharding behavior. In sharding mechanism, data is partitioned between multiple nodes which is pretty similar to horizontal partitioning technique use in parallel database systems. MongoDB has the capability to redistribute the data automatically among all the nodes when some of the nodes in the cluster have disproportionate of data as compared to other shards/ nodes across the cluster.[26]

### 2.5 RDF Linked Data/Triple Stores

RDF Linked Data/Triple store framework using DataGraft which is cloud-based platform is described below.
2.5.1 Framework

The DataGraft platform is used to serve Grafterizer as an integrated program to perform data cleaning and transformation operations. Grafterizer support to clean the tabular data and transform it into RDF. Grafterizer support two type of transformations: tabular-to-tabular and tabular-to-RDF.

Tabular-to-tabular take tabular data in CSV format and also produce transformed data in CSV format as output. While on the other hand tabular-to-RDF take input data as CSV tables and produced output RDF data in N-triples serialization format.

Grafterizer’s user interface (Figure 2.6) consists of a preview panel (on the right side) and transformation definition panel (on the left side).

In this transformation process, data is cleaned filtered and the quality of data is improved by applying different pipelines functions in Grafterizer. After cleaning tabular data, it is transformed to RDF. Besides this it is also possible to see the Clojure code generated as a result of transformation.

After improving the quality of data, the second step of transformation as mentioned above is mapping the data in RDF or Link data graph. RDF triple patterns are designed by the user whereas triples’ subjects, predicates and objects are manually specified using a mapping procedure. During this mapping process, in order to make set of triples corresponding to each data row, headers of columns are mapped to RDF nodes.

Sample of RDF mapping for CITI-SESNE dataset is shown in figure below.
Grafterizer offer to support of reusing existing RDF ontologies by providing a searchable catalog for vocabularies and also makes it possible to manage individual namespace prefixes. Each column in a dataset can be mapped as a URI node with namespace prefix assigned by user or literal node with a specified datatype. Grafterizer also support to provide handling of error when casting to datatypes.

This chapter has covered an overview and background knowledge related to storage solutions for Big data and sensor data events (IoT). The issue is highlighted that small intelligent devices (IOT) which are capable of sending data events to the server give huge rise to data volume to make it Big data. Data is mostly stored and hosted on cloud platforms and when it comes to storage cast, database solution is required to be cost effective. CITI-SENSE data is currently stored and hosted on Amazon S3 and when it comes to select an alternative database solution it is very important to keep it in mind that new proposed solution must be cost efficient.

Properties of NoSQL and traditional database systems are discussed and comparison is made to find out the pros and cons of each database. RDF/Linked data or Graph database are also discussed in this chapter as an alternative solution. DataGraft is a cloud-based platform and is used to transform, clean, map the data into RDF and publish it to DataGraft server.

From the background work and knowledge, it is observed that NoSQL databases are considered to be a good solution for sensor data (IoT) due to some unique properties, one of them include horizontal scaling and sharding which is a method for distributing data across multiple machines. MongoDB uses sharding along with high throughput operations for very large datasets which makes it a suitable database solution for large datasets[27].

<table>
<thead>
<tr>
<th>RDF Mapping</th>
<th>URI</th>
<th>ontology</th>
<th>prefix</th>
<th>namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>off_type</td>
<td>off</td>
<td>Observation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>measurement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>sensor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>sensor_time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>sensor_value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>sensor_latitude</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observed</td>
<td>v</td>
<td>sensor_altitude</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: RDF mapping in Grafterizer
3 Evaluation of existing Big Data Benchmarks

This chapter outlines and explain different Big data benchmarks currently available. The general approach is to discuss the properties of each of them and to find out if any one of them suitable and could be used to full fill the requirements stated in section1.4.1 for sensor applications in specific.

3.1 State-Of-Art (Big Data Benchmarks)

Big data involves variety of different data types that differs in volume, variety and velocity. The evolution of IoT has introduced sensory data as one of the largest and fastest Big data types. The different types of big data lead to proliferation of Big Data systems such as new generation of databases like time-series databases, data warehousing systems like Hadoop, stream processing systems like Spark, etc. Each such system aims at handling a particular/defined subset of big data management. One such category of systems that has emerged is NoSQL databases dealing with big data storage and processing.

With the advent of IoT and sensor data and eventually NoSQL databases to cope with the storage and handling of such data, comes the need for a benchmarking tool/system to evaluate different NoSQL database solutions for optimized performance. While there are several benchmarks available for assessing different Big data systems, there is the lack of sufficient tools which can evaluate the promises of high performance made by NoSQL databases when dealing with sensory Big data in specific.

Wisconsin benchmark [28] was designed for benchmarking the relational databases in particular, and is still a well-adapted benchmark for single-user evaluation of relational system.

Pavlo Benchmark [29] was developed to specifically compare the capabilities of Hadoop with those of commercial parallel Relational Database Management Systems (RDBMS). It concentrates mainly on comparison of Hadoop-based data analysis systems, using structured data sets, not practical in sensor applications domain.
TPC [30] is a non-profit corporation, has defined various database benchmarks over the time. TPC benchmarks are designed to model particular real-world applications. TPC-C and TPC-E involve transaction processing applications. TPC-D and TPC-H involves decision support systems. TPC-VMS deals with database systems hosted in virtualized environments, and TPCx-HS with profiling different Hadoop layers.

As the name suggests, TPC is focused and targeted towards transaction processing performance. NoSQL data stores do not qualify for most of TPC benchmarks, since they relax ACID transaction properties. Of all of the TPC benchmarks, TPC-DS, TPCx-BB and TPCx-HS are of interest with respect to Big Data involvement, though none of these involves sensor data for direct evaluation of NoSQL systems. TPC-DS does not benchmark operational systems and involves relational schema. It measures the performance of SQL-based Big Data Systems by expanding upon the original TPC-DS benchmark. Also, though the data generation rate can be controlled, the data updating frequency is not considered, rendering it to be a semi-controllable benchmark in terms of velocity with respect to data generation techniques TPCx-BB is an application benchmark for Big Data involving all three types of schema for structured, unstructured, and semi-structured data. But it is designed to measure the performance of only Hadoop based systems including map reduce, hive and Spark.

TPCx-HS [30] is first industry standard targeted towards Big data benchmarking, focusing mainly on Hadoop technology. It follows stepped-scale factor model. The test dataset must be chosen from the set of fixed Scale factors (1TB, 3TB, 10TB..10000TB)

BigBench [31] is a specification-based benchmark building upon existing benchmarking efforts in Big Data space such as YCSB, HiBench, Big data Benchmark, TPC-xHC, TPC-DS, GridMix, PigMix. It supports structured, semi-structured and unstructured data. The structured part of BigBench schema is adopted from TPC-DS data model depicting a product retailer., and further extended with semi- and unstructured data also. The raw data volumes can be dynamically changed based on a scale factor, but the data updating frequency is ignored in generating benchmarking data.

HiBench [32] is a big data benchmark suite that helps evaluate different big data frameworks in terms of speed, throughput and system resource utilizations. HiBench features several ready-to-use benchmarks from 4 categories: micro benchmarks, web search, machine learning, and HDFS benchmarks.
HiBench use fixed-size data as inputs also, so this system is not fully scalable when considering volume aspect of the data generation techniques. The velocity of the data is fixed and hence cannot be controlled to check performance against increase/decrease of speed with which the data is being served.

**BigDataBench** [33] is an open source Big Data benchmark suite covering five application domains; including search engine, social networks, e-commerce, multimedia data analytics and bioinformatics. The version BigDataBench 3.1 includes 14 real fully scalable data sets, types including text, graph, and table data, and eight non-scalable data sets from small seed of real data.

**Big Data Benchmark by AMPLab** [34] is the benchmark that provides comparison of five different data warehousing solutions, namely Redshift, Hive, Shark, Impala, Stinger/Tez. Each based on a different set of techniques. For example, Hive is Hadoop based while Redshift is a hosted MPP database. This benchmark measures response time via some relational queries like scans, aggregations, joins across different data sizes. The set of queries involved are chosen so that all these five systems can complete then. Also, it is run on a public cloud instead of using dedicated hardware. As opposed to Big Data Benchmark, SenseMark aims to consider NoSQL document based database systems like MongoDB, run on local hardware, with sensor benchmark data.

**JMeter** [35] Apache JMeter is an open-source java based tool used to benchmark many different technologies and services some of which include databases, web-servers, FTP-servers, web-site performance testing and many more. JMeter offers a type of testing to determine the responsiveness, reliability, scalability, throughput and interoperability of a system or any application under a provided workload.

JMeter could be used to run a single operation or millions of them simultaneously. JMeter application consist of a JMeter server that can handle and process the test plan. When a test plan is sent to JMeter server, it responds and process the plan and send the results back to host from where the test plan was sent. In this way JMeter-server has the capacity to run the test plan against a server from several client machines at the same time.

After the test plan is executed, results of the test could be collected at the master. JMetre has the capability to plot the graphs against results but for deep
visualization of the data set generated as result of the test plan could be saved in XML or CSV formats. These XML or CSV files of larger datasets could be visualized and graphs could be plotted against these datasets using different analytic tools (e.g. RStudio) for deep inspection [35].

**YCSB** (Yahoo Cloud Serving Benchmark) is a popular open-source database benchmarking tool developed by Yahoo. YCSB is capable to facilitate performance analysis of new generation database systems like NoSQL even in environments with limited resources [26]. YCSB is used to generate workload on database and write out the results as report.

YCSB is a module based and each of the database it uses has client module that is written in java. This module contains the classical CRUD operations (Create, Read, Update, Delete) that are commonly used in database systems. YCSB contain some pre-defined workloads to perform benchmarking but depending on the use cases one want to measure performance for, one can create and customize the load testing to get more accurate and realistic results for the required use case.

YCSB framework was engineered to support comparison of several NoSQL databases like Cassandra, HBase, Riak, etc. This is the one well-known benchmarking system originally designed for direct evaluation of database systems which do not support ACID. YCSB consists of a workload generator and a basic database interface, which can be easily extended to support various relational or NoSQL databases. It provides six pre-defined workloads, which simulate a cloud OLTP application (read and update operations). The reported metrics are execution time and throughput (operations per second). YCSB exercises several NoSQL solutions using a simple data model but used high performance hardware.

YCSB has been used by a number of companies [26, 36, 37] and several academic benchmarks [38, 39]. Three out of the four industry benchmarks, focus particularly on Document based database systems. Two of these benchmarks include Couchbase and MongoDB. These two benchmarks however are very narrow in the problem domain they seek to highlight by highly optimizing their studies for specific use cases on small clusters with limited variation in the type of experiments conducted. In [36] the authors modeled an interactive web application looking at a single workload comprising a 5-60-33-2% CRUD decomposition of in-memory operations only.
The third industry benchmark [26] used MongoDB and Cassandra and HBase in their experiments. This was a much more extensive benchmark than the others considering it used seven different workloads (a mixture of both customized and YCSB provided ones), on a thirty-two-node cluster, and a big data set requiring both disk-bound and memory-bound operations.

Comparison of all the big data benchmarks is shown in the table below:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Metrics</th>
<th>Data Types (structured/semi-s/un-s)</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pavlo’s Benchmark</td>
<td>Query time</td>
<td>Unstructured</td>
<td>Hive</td>
</tr>
<tr>
<td>AMP Big Data</td>
<td>Query time</td>
<td>Unstructured</td>
<td>Hive, Tez, Shark, Impala, Redshift</td>
</tr>
<tr>
<td>Benchmark TPC-H</td>
<td>Query time,</td>
<td>Unstructured</td>
<td>Hive, Pig, Impala, IBM Big SQL</td>
</tr>
<tr>
<td></td>
<td>Throughput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPC-DS</td>
<td>Query time,</td>
<td>Unstructured</td>
<td>Hive, Pig, Impala, IBM Big SQL</td>
</tr>
<tr>
<td></td>
<td>Throughput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPCx-HS</td>
<td>Performance, price</td>
<td>Structured</td>
<td>Hadoop</td>
</tr>
<tr>
<td></td>
<td>and energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BigBench</td>
<td>Query time and</td>
<td>All three</td>
<td>Teradata, Hadoop, Spark</td>
</tr>
<tr>
<td></td>
<td>BBQpH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BigDataBench</td>
<td>MultipleA</td>
<td>All three</td>
<td>Multiple technologies</td>
</tr>
<tr>
<td>HiBench</td>
<td>Execution time</td>
<td>?All three</td>
<td>Hadoop, Spark</td>
</tr>
<tr>
<td></td>
<td>and throughput</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YCSB</td>
<td>Execution time,</td>
<td>Structured</td>
<td>NoSQL databases</td>
</tr>
<tr>
<td></td>
<td>Throughput</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: State of art (Big Data Benchmarks)

None of the Big Data Benchmark cover the sensor applications in specific. The real world data sets, generated by Big Data Generator Suite, does not include sensor data with time-location significance in particular. Though there is a correlation between social networks and sensor data, but there is still difference between the basic nature of the two. With huge data streams that are non-stationary time-series data, collected at discrete intervals, there is a need to find best solution for storing and retrieving sensor data. Special attention must be paid to the ability of inserting extremely large amounts of data on a long-term basis. Tests must be performed related to both storing the data, and aggregating data under specific conditions.
3.2 Analytic tools

Different analytic tools are available to do evaluation and analysis. Tools used in this project are as follows.

RStudio

RStudio is an open source integrated development environment (IDE) for the R-programming language. It aims to simplify the development by providing syntax highlighting, editor that supports direct code execution, auto-complete and an interactive GUI for plotting, history and workspace management.

The program can run locally on the desktop (Windows, Mac, and Linux) or on a server where the GUI can be accessed through a normal web-browser. Both solutions look very similar and the most significant difference is where the processing is done. Some main features of RStudio are listed below [40].

The R programming language

R is a programming language and environment developed for performing different operations in RStudio e.g. calculation, statistical computing and plotting graphs. But using R programming language one can perform large selection of mathematical functions, statistical tools and graphic plotting [41].

Gnuplot

Gnuplot is a cross platform command line graphic utility for Linux, MS Windows, OS/2, VMS, and many other platforms that is capable to plot graphs. It uses scripts to send commands and in this way, it has capability to plot many graphs simultaneously and easily. It has the capability to take CSV files containing data to be plotted and can provide the output in many different formats including PDF, jpeg, png and direct output to LaTeX as well after plotting the graphs of the input data [42]. Gnuplot also support to provide both 2D and 3D plots.

This chapter highlighted the properties of several currently available Big data benchmarks, pros and cons of each and their comparison. Evaluation result shows that none of them could be used for sensor application in specific which is in general the requirement. So, conclusion could be made that none of these Big data benchmarks could be adopted to benchmark the performance of different database solution with respect to classical CRUD operations for sensory Big data.
4 Benchmark Requirements

This chapter highlights the requirement and general approaches being followed in order to answer the defined problem statement:

“To evaluate and benchmark different database solutions for representative sensor data for optimized performance with respect to the classical CRUD operations (Create, Read (Query), Update, Delete) with respect to response time, cost, storage capacity and scalability”.

Requirements referred to in section 1.4.1 are considered in order to propose a solution to overcome this problem statement.

4.1 Proposed Solution

After discussing the pros and cons of SQL and NoSQL database solutions, it is obvious that NoSQL databases are considered to be more efficient data stores to host sensory Big data.

The increasing use of NoSQL data stores as a solution to serve big data and lack of sufficient related benchmarks led us to conclude that there is a need for more detailed evaluations of the use of NoSQL data stores as a back-end storage for big data sensor application domain, such as SenseMark evaluating MongoDB vs WFS service calls.

To benchmark the performance of different database solutions for classical CRUD operations, there is a need to develop a benchmark tool to evaluate different database solutions for optimized performance for such type of sensory Big data. The idea here is to develop a tool to benchmark the performance for required and stressed scenarios and it must be generic enough to find out the scalability limit of each database solution for volume and velocity.

The aim is to satisfy these criteria by developing a benchmark (SenseMark) that is relevant to sensor systems, portable to different NoSQL through component programming extensibility framework, scalable to realistic data sizes, and employing simple benchmark workloads typical to sensor applications. A good benchmark must have the following characteristics listed below [43]

**Properties of a good big data benchmark**

It should be representative of real-life use cases - should generate performance insights immediately relevant to diverse and evolving big data use cases.
• It should fulfill the requirements related to data-model and database system to be benchmarked.
• It should take already developed benchmarks in consideration to build upon existing knowledge base rather than to re-invent the wheel.
• It should be scalable—should stress big data systems.
• Benchmark metrics should be of practically significant for application domain under consideration.
• It should be verifiable—results can be checked independently and successful configurations and results should be reproducible.
• Data generation should consider three V’s of big data and the significance with relation to application domain at hand. Data sets should match real-life use case under consideration. Supported schema type for generating data should coincide with schema involved in application domain used for benchmarking.
• Economical
• Fair—Data sets should be represented in realistic formats that do not inflate or deflate performance advantages for any particular system to be benchmarked.

4.1.1 SenseMark

SenseMark is the name I have given to the database benchmark that I have developed in this thesis, which will be introduced in more detail in chapter 6. SenseMark should aim at evaluating different database system alternatives, including in particular document-based NoSQL database systems, MongoDB being the database of first choice, for sensor based applications with respect to classical CRUD operations. The main purpose of this benchmark should be to assess the performance of NoSQL database systems as to find out a high-performance storage solution for sensory data.

SenseMark should be able to fit CITI-SENSE architecture and platform for data ingestion services and data publication services. SenseMark should be able to overcome performance challenges faced in Rest API and WFS Service of CITI-SENSE architecture and framework. It should be generic enough to be adopted for CITI-SESNE or any other similar sensory data architecture and framework.

This chapter highlighted the need for SenseMark - a database benchmark to evaluate the performance of different database solutions. This chapter also highlights the properties of a good Big data benchmark in order to make sure that SenseMark should be capable enough to fulfill these requirements.
5 CITI-SENSE case study

This chapter covers the case study of CITI-SENSE architecture and framework. The most common user scenarios for data publication services are evaluated and discussed in order to highlight the performance challenges faced to CITI-SENSE observatory community. (web references, http://www.citi-sense.eu/ and http://co.citi-sense.eu/).

5.1 CITI-SENSE

CITI-SENSE project is a project partially funded by the EU in collaboration with countries in Europe, Asia and Australia, and had 28 partner organizations. The project started in October 2012 and ended in October 2016. The project is meant to be driven by citizens of the respective countries, meaning that citizens will provide feedback from mobile and static sensors to make an overall environmental picture of their surroundings. Their role is to provide data from sensors they carry to give a real-time data about environment. This data ranges from air-quality, UV-radiation, pollen, indoor air-quality to noise pollution. Beside the aim of improving quality of life, the project also aims to;

1) Raise environment awareness of the citizen,
2) raise user participation in societal environmental decisions,
3)provide feedback on the impact that citizen had on decisions. [44]

There are two portals in this project. One is indoor portal and the other one is out door portal. Outdoor portal has both static and mobile sensors placed at different locations. An overview of indoor and outdoor portal is shown below[45]
Figure 5.1: CITI-SENSE Outdoor Data Portal

CITI-SENSE indoor portal is shown below [46]

<table>
<thead>
<tr>
<th>City</th>
<th>Schools</th>
<th>Sensor ID</th>
<th>Placed at current location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oslo</td>
<td>Horten Videregående skole</td>
<td>AT_5</td>
<td>04-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Horten Videregående skole</td>
<td>AT_8</td>
<td>04-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lørenskog Videregående skole</td>
<td>AT_11</td>
<td>04-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lørenskog Videregående skole</td>
<td>AT_10</td>
<td>12-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lørenskog Videregående skole</td>
<td>AT_13</td>
<td>12-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lørenskog Videregående skole</td>
<td>AT_14</td>
<td>12-11-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lambertseter Videregående skole</td>
<td>AT_6</td>
<td>01-12-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lambertseter Videregående skole</td>
<td>AT_9</td>
<td>01-12-15</td>
</tr>
<tr>
<td>Oslo</td>
<td>Lambertseter Videregående skole</td>
<td>AT_12</td>
<td>01-12-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Vodmat</td>
<td>AT_60</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Vodmat</td>
<td>AT_81</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Vodmat</td>
<td>AT_82</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Vodmat</td>
<td>AT_83</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Spodnja Šiška</td>
<td>AT_71</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Spodnja Šiška</td>
<td>AT_73</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Spodnja Šiška</td>
<td>AT_77</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>OŠ Spodnja Šiška</td>
<td>AT_78</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>Gimnazija Vič</td>
<td>AT_72</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>Gimnazija Vič</td>
<td>AT_75</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>Gimnazija Vič</td>
<td>AT_79</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Ljubljana</td>
<td>Gimnazija Vič</td>
<td>AT_80</td>
<td>24-11-15</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>Broughton Primary School</td>
<td>AT_18</td>
<td>19-02-16</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>Broughton Primary School</td>
<td>AT_23</td>
<td>19-02-16</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>Broughton Primary School</td>
<td>AT_47</td>
<td>19-02-16</td>
</tr>
</tbody>
</table>

Table 5.1: CITI-SENSE Indoor Data Portal
CITI-SENSE Citizen Observatory toolbox (COT)

COT consists of sensor units, widgets, web portals, Data, Methods involved to get information from reports, Mobile apps (City Air app+ app toolbox) a survey. This observatory toolbox provides a complete solution from capturing data, storing data, viewing data, retrieving data in different formats (CSV, JSON).

![Citizen Observatory Toolbox](image)

Figure 5.2 Citizen Observatory Toolbox[47]

CITI-sense data which is currently stored in WFS server and is available as monthly dump of CSV files as well. Each monthly dump consists of approximately 2Gb of data and monthly dumps from October, 2015 to August, 2016 are considered in this project. So, the whole data from October 2015 – August 2016 become around 25 Gb.

Data Web Services

CITI-SENSE architecture and platform provide Data Web Services consisting of CITI-SENSE data access server based on WFS-T standard. The following list shows architecture consisting of various sensor platforms and apps used to store and retrieve sensor data for volume and velocity.
A number of Web Feature Service are available to end users to connect to WFS web service and to perform different standard WFS queries encoded using OGC filter Encoding Specification (FES). The CITI-SENSE Spatial Environmental Data Server (SEDS) platform provides two main interfaces of data web services.

1. Data Ingestion Service
2. Data Publication/Access Service

5.1.1 Data Ingestion service

Data Ingestion service have been deployed on Amazon AWS and contains a single WFS endpoint which data providers utilize to register sensors and to upload sensor observations.

WFS endpoint is located at https://prod.citisense.snowflakesoftware.com/wfst and this platform provide end users with:

- GetCapabilities: provides information about the functionality that the WFS supports
- DescribeFeatureType: provides a list of feature types that the WFS offers
- GetFeature: allows the request and response of data via WFS request.
The data ingestion service is available to only few clients who are authorized to ingest data to SEDS platform to avoid unauthorized ingestion on SEDS platform. The data ingestion process consists of WFS web service allowing clients to post data by connecting to FTP server at the data provider end where sensor data is copied to SEDS platform.

5.1.2 Data publication/ Access Services

Sensor data stored in SEDS platform is accessible to users by two different web interfaces that have been realized on Amazon cloud AWS. These two web interfaces are:

1. WFS
2. REST

5.1.2.1 REST API

In CITI-SENSE for serving data a simple and more lightweight web services REST have also been deployed along WFS web services. REST interface is capable of serving data in two different formats XML and JSON.

REST service is capable of serving data for different scenarios e.g.:

- Scenario 1: Query all the observations for a specific sensor device.
- Scenario 2: Query all observations for a specific sensor device at a specific point in time (“snapshot”).
- Scenario 3: Query all observations for a specific sensor device in a given time period.
- Scenario 4: Query the latest observation for a specific sensor device.
- Scenario 5: Query the latest observation for a specific sensor device for a specific pollutant.
Figure 5.4: CTTI-SENSE Data Model Schema for WFS
Overview of Data Publication services

The tables below provide samples of the REST and WFS request patterns for the 5 scenarios.

The samples below use a sensor identifier called "CITISENSE-Test-1", which could be replaced with the sensor id for which data events are required.

<table>
<thead>
<tr>
<th>Scenario 1: Give me all observations for a specific sensor-id</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CSV:</strong></td>
</tr>
</tbody>
</table>

![Figure 5.5: All observations of sensor-id AT_5](https://prod.citisense.snowflakesoftware.com/csv/sensor/allobservations?sensorid=AT_5)

https://prod.citisense.snowflakesoftware.com/json/sensor/allobservations?sensorid=AQ_687150
Results for this query operations are shown in table 5.3.

<table>
<thead>
<tr>
<th>Sensor id</th>
<th>Start loading data events</th>
<th>Finished loading</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_5</td>
<td>4.93</td>
<td>10.36</td>
</tr>
<tr>
<td>AT_5</td>
<td>4.86</td>
<td>11.22</td>
</tr>
<tr>
<td>AT_5</td>
<td>4.75</td>
<td>10.52</td>
</tr>
<tr>
<td>AQ_687150</td>
<td>Average = 4.84</td>
<td>Average = 10.7</td>
</tr>
<tr>
<td>AQ_687150</td>
<td>2.28</td>
<td>7.98</td>
</tr>
<tr>
<td>AQ_687150</td>
<td>2.12</td>
<td>8.03</td>
</tr>
<tr>
<td>AQ_687150</td>
<td>2.03</td>
<td>7.32</td>
</tr>
<tr>
<td>AQ_687150</td>
<td>Average = 6.43</td>
<td>7.77</td>
</tr>
</tbody>
</table>

Table 5.2: Results of all observations of a specific sensor-id

This service doesn’t give all observations but only observations of couple of days which it takes randomly sometimes a couple days from March 2016 sometimes from April and so on and sometimes give time out issue.

Scenario 2: Give me all observations for a specific sensor-id at a specific point in time (time instant)

CSV:
Figure 5.7: All observations for sensor-id AT_11 at a specific point in time

Average response time = 3.47 sec

Scenario 3: Give me all observations for a specific sensor-id in a specific time period

CSV:
https://prod.citisense.snowflakesoftware.com/csv/sensor/observationfinishtime/between?sensorid=CITISENSE-Test-1&from=2001-12-17T09:00:00.000&to=2001-12-18T09:00:00.000

JSON:
https://prod.citisense.snowflakesoftware.com/json/sensor/observationfinishtime/between?sensorid=CITISENSE-Test-1&from=2001-12-17T09:00:00.000&to=2001-12-18T09:00:00.000
Figure 5.8: Observations of Sensor-id AT_11 between specified time period

https://prod.citisense.snowflakesoftware.com/json/sensor/observationfinishetime/between?sensorid=AT_10&from=2016-03-01T00:00:00.000&to=2016-03-31T00:00:00.000

Figure 5.9: All Observations of sensor-id AT_10 between specified time period

This REST call for sensor-id AT_10 for whole month of March 2016 start loading data events in 1 seconds and completely loaded results in 4.34 seconds.
Scenario 4: Give me the last observation for a specific sensor-id

**CSV:**

https://prod.citisense.snowflakesoftware.com/csv/sensor/lastobservation?sensorid=CITISENSE-Test-1

**JSON:**

https://prod.citisense.snowflakesoftware.com/json/sensor/lastobservation?sensorid=CITISENSE-Test-1

---

Figure 5.10: Last Observation of sensor-id AT_10

Average response time = 2.89 sec

---

Scenario 5: Give me the last observation for a specific sensor-id for a specific observed property (e.g. pollutant, temperature, etc.)

**CSV:**

https://test.citisense.snowflakesoftware.com/csv/sensor/lastobservation/observedproperty?sensorid=CITISENSE-Test-1&observedproperty=NO2

**JSON:**

https://test.citisense.snowflakesoftware.com/json/sensor/lastobservation/observedproperty?sensorid=CITISENSE-Test-1&observedproperty=NO2

https://test.citisense.snowflakesoftware.com/json/sensor/lastobservation/observedproperty?sensorid=AQ_687150&observedproperty=NO
This REST call gives time out issue but in principal this sensor id has recorded NO values which could be seen the following REST call

https://prod.citisense.snowflakesoftware.com/json/sensor/allobservations?sensorid=AQ_687150

Performance Challenges in REST Calls

A particular issue is with the REST service for CSV which don't works and gives timeout error. This REST service only works for JSON in some situations. Some examples of REST service with performance challenges are mentioned below.

https://prod.citisense.snowflakesoftware.com/csv/sensor/observationfinishtime/between?sensorid=AT_8&from=2016-03-01T00:00.000&to=2016-03-02T01:39:00.000

Performance challenges for various calls and typically get time out for larger CSV calls. While trying to get CSV data even for one day, it gives time out error. I managed to get CSV data for maximum 8 hours’ time period. Example REST service is listed below.

https://prod.citisense.snowflakesoftware.com/csv/sensor/observationfinishtime/between?sensorid=AT_10&from=2016-03-01T00:00.000&to=2016-03-01T08:00:00.000

There are performance issues with JSON as well. Here is another example for a specific sensor id between specific time period from October 10th, 2015 to September 14th, 2016 time.

https://prod.citisense.snowflakesoftware.com/json/sensor/observationfinishtime/between?sensorid=AT_77&from=2015-10-10T00:00.000&to=2016-09-14T00:00:00.000

This rest service only gives observations from 09.02.2016 onwards but there are observations in Oct 2015 for the same sensor id as well for example the service below gives results from 10.10.2015 - 14.10.2015.
So, in this last REST call we can see there are recorded measurements in the server which were not published by the previous REST call of time stamp 10.10.2015 – 14.09.2016.

5.1.2.2 WFS (Web feature Service)

There is various numbers of WFS service endpoints available and after getting connected to these WFS web service, end-users can perform standard WFS queries using OGC Filter Encoding Specification (FES).

There are two distinct end points for the WFS:

<table>
<thead>
<tr>
<th>WFS End Points</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://prod.citisense.snowflakesoftware.com/wfs">https://prod.citisense.snowflakesoftware.com/wfs</a></td>
<td>General, all purpose, WFS endpoint</td>
</tr>
<tr>
<td><a href="https://prod.citisense.snowflakesoftware.com/wfslr">https://prod.citisense.snowflakesoftware.com/wfslr</a></td>
<td>Endpoint specifically for accessing the latest observation.</td>
</tr>
</tbody>
</table>

Different Scenarios offered by WFS service are mentioned below:

<table>
<thead>
<tr>
<th>Scenario 1: Give me all observations for a specific sensor-id</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XML</strong></td>
</tr>
</tbody>
</table>

```xml
  <PropertyIsEqualTo> 
  <ValueReference>cts:sensorID/@xlink:href</ValueReference> 
  <Literal>CITISENSE-Test-1</Literal> 
  </PropertyIsEqualTo> 
  </Filter>
```
Scenario 2: Give me all observations for a specific sensor-id at a specific point in time (time instant)

XML

<And>
  <PropertyIsEqualTo>
    <ValueReference>cts:sensorID/@xlink:href</ValueReference>
    <Literal>CITISENSE-Test-1</Literal>
  </PropertyIsEqualTo>
  <PropertyIsEqualTo>
    <ValueReference>cts:finishtime</ValueReference>
    <Literal>2001-12-18T10:30:47.000</Literal>
  </PropertyIsEqualTo>
</And>
</Filter>

Scenario 3: Give me all observations for a specific sensor-id in a specific time period

XML

<And>
  <PropertyIsEqualTo>
    <ValueReference>cts:sensorID/@xlink:href</ValueReference>
    <Literal>CITISENSE-Test-1</Literal>
  </PropertyIsEqualTo>
  <PropertyIsBetween>
    <ValueReference>cts:finishtime</ValueReference>
    <LowerBoundary>
      <Literal>2001-12-18T09:00:00.000</Literal>
    </LowerBoundary>
    <UpperBoundary>
      <Literal>2001-12-18T09:00:00.000</Literal>
    </UpperBoundary>
  </PropertyIsBetween>
</And>
</Filter>
### Scenario 4: Give me the last observation for a specific sensor-id

**XML**

https://prod.citisense.snowflakesoftware.com/wfs?service=wfs
& version=2.0.0
& request=GetFeature&
& typename=cts:Observation
& filter=
    <PropertyIsEqualTo>
      <ValueReference>cts:sensorID/@xlink:href</ValueReference>
      <Literal>CITISENSE-Test-1</Literal>
    </PropertyIsEqualTo>
  </Filter>

### Scenario 5: Give me the last observation for a specific sensor-id for a specific observed property (eg. pollutant, temperature, etc.)

**XML**

https://prod.citisense.snowflakesoftware.com/wfs?service=wfs
& version=2.0.0
& request=GetFeature&
& typename=cts:Observation
& filter=
    <And>
      <PropertyIsEqualTo>
        <ValueReference>cts:sensorID/@xlink:href</ValueReference>
        <Literal>CITISENSE-Test-1</Literal>
      </PropertyIsEqualTo>
      <PropertyIsEqualTo>
        <ValueReference>//cts:observedProperty</ValueReference>
        <Literal>NO2</Literal>
      </PropertyIsEqualTo>
    </And>
  </Filter>
Oslo 24 hours WFS service and results.

Figure 5.11: WFS service for OSLO 24 hours’ data

Figure 5.12: Oslo 24 hours

Figure 5.13: Time out error while calling Oslo 24 hours’ measurements
Oslo 1 hour WFS Service

Figure 5.14: Oslo one hour WFS service facing time out issue
Performance Challenges in WFS Service

There are performance challenges faced in WFS service as well. It doesn’t provide exact results. Sometimes it takes a long time to start loading data and sometimes the service gives time out error messages. The WFS service is also not stable to retrieve data from the server for specific sensor id in specific time period or all observation for a specific sensor id. Sometime it returns measurements out of the defined boundary box and sometimes it returns time out issue.

This chapter described all the services available to user groups in the CITI-SENSE platform and architecture. Performance challenges are discussed and highlighted for both data ingestion service and data publication service. Data publication service consist of WFS and REST calls and both of these service shows some performance challenges which are evaluated and highlighted in this chapter.

The short term solution that was made in order to handle the CITI-SENSE server performance issue was to always keep a separate cache of the data from the latest hour and to serve this – as the most important user query was the query about the latest data.
II
SenseMark - Concept, Design & Implementation
6 SenseMark – Concept and Design

This chapter provides an overview of the concept and design, requirements from the proposed benchmark (SenseMark). It covers target scenarios of CITI-SENSE that is the requirement and later analyzed the stressed scenarios to find the scalability limit of the proposed database solution. Design of SenseMark is discussed with respect to the requirements, technical challenges that need to be fulfilled by SenseMark are discussed in this chapter.

6.1 Concept of SenseMark

SenseMark aims to provide a sensing architecture prototype that must be capable of supporting large scale sensory inputs and provide a scalable storage solution for such type of sensor data events used in CITI-SENSE. SenseMark aims to perform benchmarking for classical CRUD (Create, Read, Update, Delete) operations for such type of sensor data events captured by sensor devices (IoT). Two type of scenarios are created and tested to benchmark the performance of database solution.

Synthetic sensor data events are generated, evaluated and their performance is analyzed for classical CRUD operations against WFS-service calls and MongoDB as a representative NoSQL database system. The workloads are configurable and the benchmark demonstrates how different configuration settings affect the performance of database system. The query workload attempts to include: (1) queries that are most common and useful for real-world sensor location applications, and (2) queries that are simple but can serve as basic building blocks for more complex queries.

6.1.1 Target Scenario (Requirement)

Target scenario is required to support CITI-SENSE project which has approximately 2000 sensors placed in 10 different locations and each sensor is sending measurement potentially up to once in a second which means that each city has 200 sensors sending data events to database each second. This implies that database is required to store 2000 events/second in total and respond to the read (Query) operations involved in CITI-SENSE. So, in the first part benchmarking is done for classical CRUD operations using SenseMark to support 2000 sensors sending data at a rate of 2000 events/second. In the current project implementation some of the sensor data is batched up in files that are ingested into the CITI-SENSE server once every 5 minutes, 15 minutes or hour.
6.1.2 **Stressed Scenario**

In the second part a stressed scenario is made and considered in order to test the scalability of storage solution and also to make sure required target scenario is achieved and proposed database solution fulfill the requirement. For this purpose, total no. of sensors is first increased to 4000 (twice) and benchmarked, then increased to 10000 (five times) and in the end increased to 20000(10 times) to see that how much scalability proposed database solution can offer for such type of sensor data events for classical CRUD operations. In each load test, 60 million events are generated and stored in SenseMark-storage solution.

6.1.3 **Benchmark operations to be considered**

Benchmarking is done using SenseMark for classical CRUD operations and results are analyzed to compare the performance. Following are the operations considered and detailed scenarios in each of the operations.

1. Read Performance benchmark
2. Write performance benchmark

6.1.3.1 **Read Performance benchmark**

In this context, most common scenarios used in WFS service are considered to perform standard read operations for such type of sensor data events involved in CITI-SENSE.

- Scenario 1: Give me all observations for a specific sensor-id
- Scenario 2: Give me all observations for a specific sensor-id at a specific point in time (time instant)
- Scenario 3: Give me all observations for a specific sensor-id in a specific time period
- Scenario 4: Give me the last observation for a specific sensor-id
- Scenario 5: Give me the last observation for a specific sensor-id for a specific observed property (e.g. NO, CO, etc.)

6.1.3.2 **Write performance benchmark**

Write performance benchmarking is done using target scenario for CITI-SENSE and also for stressed scenario (described earlier) to see the scalability limit of proposed database storage solution. To fulfill target scenario, proposed database solution is required to store 2000 events/second. Stressed scenario is evaluated and tested by increasing the no. of sensors twice, 5 times and 10 times respectively.
This is real time data generation so each load test run for 600 seconds (10 mins) to generate and store data in database.

6.2 Design of SenseMark

First step include to develop the storage component responsible for storing sensor data events (external data) and system component events (internal data) in Storage layer of sensing architecture[48]. The next step is to provide the Adapter library (Sensing layer) consisting of Java library for developing adapters capable of receiving data from all sensor types. Conceptual sensing architecture is shown in the figure 6.1.

![Figure 6.1: Conceptual Architecture of SenseMark](image)

6.2.1 Requirements for SenseMark Architecture

SenseMark sensing architecture is supposed to be capable of providing best solution for sensing data events on both Sensing layer and Storage layer. SenseMark sensing architecture must be capable to cover a set of non-functional requirements e.g. performance, reliability, scalability, usability, security, maintainability, testability, configurability and reusability. Non-functional requirements from sensing and storage layer are shown in table 6.1.
<table>
<thead>
<tr>
<th>Non-functional requirements</th>
<th>Storage Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Must provide performance in line for volume, velocity and variability requirements of other components.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Must be capable of scaling proportionally with the data volume and velocity.</td>
</tr>
<tr>
<td>Usability</td>
<td>n/a</td>
</tr>
<tr>
<td>Reliability</td>
<td>It is the most important component and must be highly reliable without any chance of error.</td>
</tr>
<tr>
<td>Security</td>
<td>user access and component access control restricting its usage to approved CITI-SESNE users/components will need to be addressed.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>n/a</td>
</tr>
<tr>
<td>Testability</td>
<td>requires testability to ensure the correctness of the storage, interfaces.</td>
</tr>
<tr>
<td>Reusability</td>
<td>must be generic/reusable</td>
</tr>
<tr>
<td>Configurability</td>
<td>Must be capable to be configured and run on different operating systems</td>
</tr>
</tbody>
</table>

Table 6.1: Non-functional requirements from sensing and storage layer

6.2.2 Technical Challenges

The technical challenges for sensing and storage layers are major issues related to big data aspects i.e.

- volume
- velocity
- variety

Volume

Sensing and Storage layer must be capable of storing and processing larger amount of sensor data events coming from hundreds of business sensors continuously and rapidly. To manage this larger and rapidly increasing volume of data is a big challenge because data volumes are sometimes increasing faster as compared to CPU speed and other computing resources. In ideal case sensing and storage layer must be capable to scale automatically according to the requirements for larger volumes of sensor data events because hundreds of sensors may be used to send data events.
Velocity

Velocity is another major aspect for sensing architecture when it comes to speed of moving data events and reaction time. Data transfer rate of sensors in business use cases varies from typically 60 seconds down to 20ms. The no. of data processed in the proposed sensing architecture will be measured in no. of events per second. In initial typical scenario, SenseMark target to 1000 sensors at 20ms which implies sensing and storage layer must be capable to process 50 000 sensors data events per second.

Variety

Sensing and storage layers must be capable to process and store data coming from different sensor types and platforms. In other words, sensing architecture must be capable to support heterogeneous types of data.

6.2.3 Proposed Approach

Sensing layer

The Sensing layer is capable to provide data acquisition from external sensor sources into the system.

The proposed approach in sensing architecture is that Storage Registry Service which provides a RESTful Registry API that allows registration of sensors, their metadata and updating and querying metadata. This sensor registry information is stored in Sensor Registry database. Adopters/ Sensing layer provide data acquisition for retrieving sensor data events from different sensor types. Components of sensing architecture are shown in figure 6.2.
Storage layer

The storage layer is capable to provide storage, query, batch processing for the sensing architecture. The storage layer is capable to store data events coming from different sensor devices and also to be able to process standard query operations for such type of sensor data events. Thus, the main components of the storage layer are as follows:

- **Storage Writer Service** is capable to monitor the data events that are processed and published on the system.
- **Storage Reader Service** provide RESTful API to query the stored data events.

Standard Queries

SenseMark storage must be capable to provide most commonly used standard read / query operations for such type of sensor data events such as data events for a specific sensor device at a specific location and specific period of time etc.

The SenseMark concept and design is successfully evaluated in this chapter, target scenario from CITI-SENSE project is highlighted and stressed scenarios are also created to test the scalability limit of the database solution for read-write operations. Technical challenges to be covered by the SenseMark architecture are also identified. Moreover, the component of sensing architecture is also identified and designed to evaluate the performance of the proposed database.
7 SenseMark – Implemented and Analyzed for MongoDB

In this chapter, implementation of SenseMark on MongoDB is deployed to benchmark the performance for classical CRUD operations. Write performance is first benchmarked for CITI-SENSE target scenario that writes 2000 events/sec on MongoDB. Later, stress testing is performed by gradually increasing the total number of events/sec. Read performance is also measured for typical CITI-SENSE scenarios that are mentioned in section 6.1.3.1.

7.1 Implementation

I have decided to first implement SenseMark using the document store MongoDB. MongoDB stores data as document which in this context is a schema-less data model because the attributes are not defined and wide ranges of values are permitted. MongoDB and CouchDB are the two most popular document stores available. The main reason behind selecting MongoDB to implement SenseMark are as follows.

1. Data Model suitability
2. Performance and feature set

Data Model suitability

The storage requirement for CITI-SENSE data events require a schema-less data model that can have different events e.g. simple, derived predicted, questionnaire, response etc. These events can have arbitrary no. of properties and values which also very important fact to be considered. MongoDB data model which is schema-less and provide flexibility and heterogeneous document collections. MongoDB collections are capable of storing hundred thousand different structure documents and also it offers flexibility in terms of storage capabilities specially in a research prototyping environment.

Performance and feature set

MongoDB and CouchDB are two main and most popular document stores available now a day. MongoDB is used widely now a day due to performance improvements in latest released version 3.2. MongoDB offers a variety of feature sets for example when it comes to batch processing capabilities, supporting aggregation pipeline and Hadoop style map reduce functions.
MongoDB is installed locally on a laptop computer and also laptop hard-drive is used as storage device to host data. The technical specifications of the system used for benchmarking are as follows:

**System Specifications**
- Hp EliteBook 8540w
- CPU: Intel(R) Core i7 M620 @ 2.67 GHz
- Cache: L2CacheSize 256Kb, L3CacheSize 4 MB
- Installed Memory (RAM): 8.00 GB
- System type: 64-bit Operating System
- HDD 768 GB (Approximately 70 GB free storage between each benchmark execution)

Data Generation SenseMark is executed and run on IntelliJ IDEA. Stored data events are verified and query operations are performed using MongoDB management tool MongoChef.

### 7.2 Benchmarking MongoDB

The benchmark tool is first implemented and analyzed for MongoDB according to the scenarios mentioned in design part.

#### 7.2.1 MongoDB Write Performance Benchmark

Write Performance benchmark is done for real and stressed scenarios. Real scenario is the requirement for database solution from CITI-SENSE project to be fulfilled and then stressed scenarios are performed to see the scalability of MongoDB until it crashes. In stressed scenarios, total no. of sensors is increased gradually to find out the scalability limit of MongoDB. Each test is performed several times to avoid any system error that could affect the performance of database while writing data events.

#### 7.2.1.1 CITI-SENSE Scenario benchmark

Write performance benchmark is first implemented for actual scenario of CITI-SENSE in which there are approximately 2000 sensors each sending data event once in a second. Write performance benchmarking is done for writing 12 00 000 events in 600 seconds. This is a real-time data generation so I selected 600 seconds for event writes. Tables 6.1 describes the details of real scenario to be tested.
Table 7.1: MongoDB write performance test for CITI-SENSE real scenario

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of sensors</td>
<td>2000</td>
</tr>
<tr>
<td>Data events frequency for each sensor</td>
<td>1 second</td>
</tr>
<tr>
<td>Total no. of messages sent in 5 second</td>
<td>10000</td>
</tr>
<tr>
<td>Run time</td>
<td>600 seconds</td>
</tr>
<tr>
<td>Total no. of events generated and stored in 600 sec</td>
<td>12,000</td>
</tr>
</tbody>
</table>

Sensor data events are generated for 2000 and sent at rate of 1000 events/second to MongoDB server and write operation performance is tested. Average write events/second is shown in figure below.

![Figure 7.1: Average write events of 2000 sensors sending 2000 events/s](image)

The test is performed on local machine described earlier and same scenario is tested 3 times to attain accuracy and the results are compared. Boxplot for 3 tests performed at same machine with same criteria of write operations is shown in figure below.
It could be observed that MongoDB shows linearity when 2000 sensors are sending data event per second. Each test is performed three times to achieve accuracy and to avoid possible system error that could cause problems in write operations. Comparison of three tests performed is shown below:

<table>
<thead>
<tr>
<th>Total no. of writes (2000/ sec)</th>
<th>Average writes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>1940.71</td>
</tr>
<tr>
<td>Test 2</td>
<td>1933.24</td>
</tr>
<tr>
<td>Test 3</td>
<td>1942.05</td>
</tr>
</tbody>
</table>

### 7.2.1.2 Stressed Scenarios Benchmarking of MongoDB

Stressed scenarios are created to test the scalability of MongoDB for such type of sensor data events. To find the scalability limit of MongoDB, benchmarking is performed by increasing the total no. of sensors gradually and to see that where MongoDB crashes.
MongoDB Scalability test for 5000 sensors

A scenario is created in which there are 5000 sensors and each of them is sending sensor data event once in a second. Benchmarking parameters are shown in table below:

<table>
<thead>
<tr>
<th>No. of sensors</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data events frequency for each sensor</td>
<td>1 second</td>
</tr>
<tr>
<td>Run time</td>
<td>600 seconds</td>
</tr>
<tr>
<td>Total no. of events generated and stored in 600 sec</td>
<td>300 000</td>
</tr>
</tbody>
</table>

Table 7.2: MongoDB write performance test criteria for 5000 sensors

![Average write events of 5000 sensors sending 5000 events/s](image)

Figure 7.3: Average write events of 5000 sensors sending 5000 events/s

Average writes / second= 4628 and again test is performed three times to avoid system error and to achieve accuracy. The boxplot for three tests is shown below.
It could be seen in the box plot that average writes/second are almost same in all the tests but in the third test, write events are scattered widely. The reason could be some work load on the system in parallel while write operation is performed or some back-ground process working in parallel.

<table>
<thead>
<tr>
<th>Total no. of writes (5000/sec)</th>
<th>Average writes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>4628.84</td>
</tr>
<tr>
<td>Test 2</td>
<td>4615.92</td>
</tr>
<tr>
<td>Test 3</td>
<td>4814.69</td>
</tr>
</tbody>
</table>

**MongoDB Scalability test for 10000 sensors**

Now the performance of MongoDB is tested for 10000 sensors each sending data event once per second. Benchmark parameters are shown in table below:

<table>
<thead>
<tr>
<th>No. of sensors</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data events frequency for each sensor</td>
<td>1 second</td>
</tr>
<tr>
<td>Run time</td>
<td>600 seconds</td>
</tr>
<tr>
<td>Total no. of events generated and stored in 600 sec</td>
<td>60 00 000</td>
</tr>
</tbody>
</table>

Table 7.3: MongoDB write performance test criteria for 10000 sensors
Figure 7.5: Average write events of 10000 sensors sending 10000 events/s

Average writes / second= 8783 and again test is performed three times to avoid system error and to achieve accuracy. The boxplot for three tests is shown below.

Figure 7.6: 10000 write events/sec performed 3 times
Average write events in three tests are shown below.

<table>
<thead>
<tr>
<th>Total no. of writes (10000/ sec)</th>
<th>Average writes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>8783.64</td>
</tr>
<tr>
<td>Test 2</td>
<td>8961.64</td>
</tr>
<tr>
<td>Test 3</td>
<td>8988.305</td>
</tr>
</tbody>
</table>

**MongoDB Scalability test for 20000 sensors**

In this load testing, 20000 sensors are selected to send 20000 data events/sec to MongoDB. Selected parameters are shown below

<table>
<thead>
<tr>
<th>No. of sensors</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data events frequency for each sensor</td>
<td>1 second</td>
</tr>
<tr>
<td>Run time</td>
<td>180 seconds</td>
</tr>
<tr>
<td>Total no. of events generated and stored in 600 sec</td>
<td>36 00 000</td>
</tr>
</tbody>
</table>

In this case MongoDB managed to store average of 15800 events/sec. Performance is plotted in figure below:

![Figure 7.7: Average write events of 20000 sensors sending 20000 events/s](image-url)
Again, test with same benchmarking parameters is repeated three times to avoid the system error and to achieve accuracy and it is observed that in every test MongoDB shows almost same performance and average write events/sec are between 15500 – 16000 events/second. Boxplot for three tests is shown in figure below.

![Boxplot View](image)

Figure 7.8: 20000 write events/sec performed 3 times

Average write data events stored by MongoDB are as follows.

<table>
<thead>
<tr>
<th>Total no. of writes (20000/ sec)</th>
<th>Average writes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>15760</td>
</tr>
<tr>
<td>Test 2</td>
<td>15453</td>
</tr>
<tr>
<td>Test 3</td>
<td>16039</td>
</tr>
</tbody>
</table>

**MongoDB Scalability test for 50000 sensors**

Now the performance of MongoDB is tested for 50000 sensors each sending data event once per second. Benchmark parameters are shown in table below:
<table>
<thead>
<tr>
<th>No. of sensors</th>
<th>50000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data events frequency for each sensor</td>
<td>1 second</td>
</tr>
<tr>
<td>Run time</td>
<td>600 seconds</td>
</tr>
<tr>
<td>Total no. of events generated and stored in 600 sec</td>
<td>30 000 000</td>
</tr>
</tbody>
</table>

As in this load testing, 500000 data events are send to MongoDB but MongoDB managed to store an average data events of about 150000 events/sec and also it crashed as well after storing 1 000 000 events. My system didn’t support to process this huge amount of data. I tried load testing for 50000 sensors but every time my system crashed and stopped storing data after some time. Boxplot of 50000 sensor data events sent to MongoDB per second is shown in figure below.

![Boxplot of 50000 sensor data events sent to MongoDB per second](image)

**Figure 7.9**: Average write events of 50000 sensors sending 50000 events/s

It could be observed that MongoDB failed to store this amount of generated data events and average writes in this case are only 13524 writes/sec. In this load testing Disk usage of the system was 100% for a long time and RAM usage was also above 90% and system crashed.
7.2.2 **MongoDB Read Operations Performance Benchmark**

In this context, the most common scenarios used in the WFS service are considered to perform standard read operations for such type of sensor data events stored in MongoDB. Load testing is performed to populate the database and now query operations will be performed to retrieve the data back by query operations performed in the WFS service.

7.2.2.1 **Scenario 1: Give me all observations for a specific sensor-id**

- show dbs
- `db.SensorDataEvents.find().pretty()`
- `db.SensorDataEvents.find( {"sensorId": "AT_10"} ).pretty()`

![MongoDB Command Line Interface](image)

**Figure 7.10:** All Observations of a specific sensorId

It could be observed that this sensor is sending observation every second and this query returned all observations of sensor id AT_10 which are more than 5 million in the database are filtered and loaded from database within 0.215 sec.

7.2.2.2 **Scenario 2: Give me all observations for a specific location**

- `db.SensorDataEvents.find( {"Location": "Oslo" } ).pretty()`
- `db.SensorDataEvents.find( {"Location": "Edinburgh" } ).pretty()`
7.2.2.3 Scenario 3: Give me all observations for a specific location in a specific time period

- `db.SensorDataEvents.find({ "Location": "Oslo", timestamp: { $gte:ISODate("2016-02-08T00:00:00Z"), $lt: ISODate("2016-09-20T00:00:00Z") } }).pretty()`
7.2.2.4 Scenario 4: Give me all observations for a specific sensor-id at a specific point in time (time instant)

- `db.SensorDataEvents.find({ "sensorId": "AQ_10", timestamp: {$gte:ISODate("2016-03-01T10:00Z")})`
7.2.2.5 Scenario 5: Give me all observations in a specific time period

- db.SensorDataEvents.find({ timestamp: { $gte: ISODate("2016-02-01T11:00:09Z"), $lt: ISODate("2016-02-09T11:10:0Z") } })

Figure 7.11: All observations in a specific time period

This query returned observations from 01.02.2016 – 09.02.2016 in 5.693 sec. Observations in this time stamp are estimated as follows

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of sensors (AT_1-100)</td>
<td>100</td>
</tr>
<tr>
<td>No. of events/ sec by one sensor</td>
<td>1</td>
</tr>
<tr>
<td>Total no. of events/sec by 100 sensors</td>
<td>100</td>
</tr>
<tr>
<td>Total data events in one day</td>
<td>100(60<em>60</em>24) = 8 640 000</td>
</tr>
<tr>
<td>Total data events in 9 day</td>
<td>77 760 000</td>
</tr>
</tbody>
</table>

Table 7.4: Total no. of 9 days’ data events in MongoDB

- db.SensorDataEvents.find({ timestamp: { $gte: ISODate("2016-07-01T00:00:00Z"), $lt: ISODate("2016-10-02T00:00:00Z") } }).pretty()
### 7.2.2.6 Scenario 6: Give me all observations for a specific sensor-id in a specific time period

- `db.SensorDataEvents.find({ "sensorId": "AE_48", timestamp: { $gte: new Date("2016-07-01T00:00:00Z"), $lt: new Date("2016-10-02T00:00:00Z") } }).pretty()`

![Figure 7.12: All observations for a specific sensor-id in a specific time period](image)

Response time is this query is 1.990 sec
7.2.2.7 Scenario 7: Give me the last observation for a specific sensor-id

- db.SensorDataEvents.find({ "sensorId": "AB_49" }).sort({$natural:-1}).limit(1)

Response time = 2.07 sec

7.2.2.8 Scenario 8: Give me the last observation for a specific sensor-id for a specific observed property (e.g. NO, CO, etc.)

- db.getCollection('SensorDataEvents').find({"sensorId": "AE_10","info.measured_property" : "CO"}).sort({$natural:-1}).limit(1)
This chapter covered the implementation of the SenseMark on MongoDB that was successfully benchmarked for read-write operations. Furthermore, write operations were implemented for stress testing to find the scalability limit of the MongoDB.

Response time = 0.003 sec
8 SenseMark – Considered for RDF/Linked Data

This chapter considers the SenseMark for linked data RDF/graph database solution. This chapter also covers the RDF mapping of CITI-SENSE data and publish it to DataGraft server to analyze and benchmark the performance of graph database solution.

8.1 DataGraft (RDF/Linked Data)

DataGraft platform is used to serve Grafterizer as an integrated program to perform data cleaning and transformation operations. Grafterizer support to clean the tabular data and transform it into RDF. Grafterizer support two type of transformations: tabular-to-tabular and tabular-to-RDF.

CITI-SENSE monthly dumps are available as CSV files. Each monthly dump consists of eight different CSV files named as answer, measurement, observation, pilotcity, question, questionnaire, response, sensordevice, sensorprovider. Size of all 8 files is about 1.78GB out of which measurement folder occupy most of the space around 1.5GB.

Data transformation is performed on each of the CSV dataset using Grafterizer. Pipeline functions are performed to filter rows and columns and to remove unnecessary columns in the datasets and also those which are empty and have record stored. Grafterizer’s user interface shown in figure below consist of a preview panel on the right and transformation definition panel on the left side of Grafterizer GUI. One of a pipeline operation performed on CITI-SENSE CSV data is shown below.

![Figure 8.1: Grafterizer GUI in DataGraft](image)
After data transformation and cleaning, CSV data set from CITI-SENSE monthly dumps are mapped to RDF. RDF triple patterns are designed by the user whereas triples' subjects, predicates and objects are manually specified using a mapping procedure. During this mapping process, in order to make set of triples corresponding to each data row, headers of columns are mapped to RDF nodes. One of the RDF mapping done for CITI-SENSE CSV data set is shown in figure below:

![RDF Mapping of CITI-SENSE CSV](image1)

**Figure 8.2: RDF Mapping of CITI-SENSE CSV**

![Overview of Clojure Code](image2)

**Figure 8.3: Overview of Clojure Code**
For doing mapping larger CSV files to RDF, a Java based executable is downloaded in the system and mapping is performed locally on the machine to transform the data into RDF.

Figure 8.4: Downloadable executable in DataGraft

After executing the transformation on the data set and retrieving the results, the obtained RDF file is shown in figure below:

Figure 8.5: Sample of RDF file

Then each of the RDF mapped files is published as a page on the DataGraft server and SPARQL queries are performed to retrieve back the data. Sample SPARQL query is shown in figure below:
8.2 Performance Challenges with DataGraft / RDF

There are some performance challenges faced while mapping the CITI-SESNE monthly dumps of tabular data into RDF. The most important of which is with the scalability issue with the DataGraft and the database server.

The most important issue is with larger amount of storage data emerging as a result of RDF mapping. Each monthly dump of CITI-SENSE consists of approximately 1.78Gb of data and when mapped to RDF it becomes almost 12-15 times larger as compared to original CSV data set. In CITI-SENSE use case where we have about 22GB of data for 12 months, when mapped to RDF it would become more than 300Gb and the storage cost will increase a lot. It would not be a good solution to map such type of sensor data into RDF which is increasing gradually and giving huge rise to data volume hosted on cloud platform. Another performance challenge faced while mapping the CITI-SENSE data into RDF is that DataGraft doesn’t support online RDF mapping of CSV data sets whose size is greater than 10Mb. In such case, a local RDF mapping could be done using downloadable executable. Moreover, local Java transformation also support CSV datasets upto maximum size of 100Mb and in CITI-SESNE monthly dumps, measurement CSV files size is about 1.5Gb in size and it is not possible to map it directly even with local java transformation. It is required to break CSV dataset into several CSV files of size 100Mb that itself is a challenge and time consuming process. After splitting the larger CSV files into smaller one’s of size 100Mb, when performing transformation and mapping into RDF using downloadable executable, it takes a lot of time to do mapping for each dataset. After mapping 100Mb of CSV
into RDF it becomes 1500MB of RDF dataset which is again a challenge to store and publish it on the DataGraft server. Besides this, due to some scalability limit and technical issues it was not possible to move and store 1500Mb of RDF to DataGraft server directly from DataGraft.

This chapter covered mapping of CITI-SENSE sensor data into RDF/Linked data and published it to DataGraft. Also, it covered some data cleaning and data transformation operations as well. Results show that when CSV dataset is mapped into RDF it becomes 15 times larger as compared to the original dataset. Due to these performance challenges and scalability issues, it was decided not to use graph database for such type of sensor data that is increasing gradually.

The conclusion of the evaluation of SenseMark for linked data RDF/graph database solution with DataGraft is that the current version of DataGraft did not support well the data volumes of sensor data that was targeted in SenseMark. Also the overhead in space usage with a factor of more than 10 when converting the sensor data from CSV representation to RDF/Linked data representation led to the conclusion that the advantage of using RDF/Linked data would not be measured in times of higher performance. The RDF/Linked data approach is, however, an interesting solution when data from many different representations and data stores needs to be integrated, but not necessarily for handling large data volumes of structured sensor data.

The identified scalability issues in DataGraft by SenseMark is now being addressed by the DataGraft team – but it was concluded that it was not relevant to complete the full SenseMark benchmark for DataGraft and RDF/Linked data following this experimentation and analysis.
III

EVALUATION
9 Evaluation and Results

This chapter covers the evaluation and results obtained by implementing SenseMark on MongoDB and compare them with other databases, e.g. WFS service and graph database (RDF/Linked data).

9.1 Evaluation of SenseMark

First of all, SenseMark results are evaluated and compared with the properties of a good Big data benchmark as described in chapter 4. As a result of the evaluation, it is observed that the SenseMark implementation on MongoDB fulfills most of the suggested properties of a good Big data benchmark.

<table>
<thead>
<tr>
<th>Suggested properties of a good Big Data benchmark</th>
<th>SenseMark Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>It should fulfill the requirements related to data-model and database system to be benchmarked</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>It should take already developed benchmarks in consideration to build upon existing knowledge base rather than to re-invent the wheel.</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>It should be scalable</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Benchmark metrics should be of practically significant for application domain under consideration.</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>It should be verifiable results can be checked independently and successful configurations and results should be reproduce-able.</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Economical</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Data generation should consider three V’s of big data and the significance with relation to application domain at hand. Data sets should match real-life use case under consideration. Supported schema type for generating data should coincide with schema involved in application domain used for benchmarking.</td>
<td>Partially fulfilled</td>
</tr>
<tr>
<td>Fair Data sets should be represented in realistic formats that do not inflate or deflate performance advantages for any particular system to be benchmarked.</td>
<td>Fulfilled</td>
</tr>
</tbody>
</table>

Table 9.1: Comparison of SenseMark with an ideal Big data benchmark

As shown in the table 9.1, comparing SenseMark with the properties of an ideal benchmark, it is observed that SenseMark meet all the promises set for a good big data benchmark. When we look at a good Big data benchmark property which is Data generation should consider three V’s of big data and the significance with relation to application domain at hand. Data sets should match real-life use case under consideration, it is observed that SenseMark partially fulfilled this property.
The reason is that while generating data, SenseMark is not dealing with all big data aspects which are volume, velocity and variety. SenseMark is generic enough to be used according to the requirements. As exhibited in the table, our developed benchmark has been able to fulfill the data-model and database system requirements. It has also accepted other developed benchmarks into consideration. Our work has also shown the scalability and reliability of the proposed benchmark (SenseMark). The introduced benchmark is able to verify and reuse the results that are obtained. It has been observed that the data generation is able to consider three V’s of the big data. Furthermore, the data sets can be matched with the real time use cases. The supported schema can also coincide with the schema that is involved in application domain that is used to benchmark. Finally, we also tested that the fair data sets can be represented in a realistic format without inflating or deflating performance advantages.

SenseMark is evaluated with CITI-SENSE perspective and to figure out that if it fits and meets the CITI-SENSE requirements from section 6.1.1. An evaluation is shown in table below.

<table>
<thead>
<tr>
<th>CITI-SENSE Requirement</th>
<th>SenseMark Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Velocity</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Variety</td>
<td>Partially Fulfilled</td>
</tr>
<tr>
<td>Target Scenario (Requirement) 2000 events/sec</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Stressed Scenario Tests</td>
<td>Fulfilled</td>
</tr>
</tbody>
</table>

Table 9.2: Evaluating SenseMark for CITI-SENSE requirements

As displayed in the table 9.2, the developed benchmark can sufficiently fulfill the requirements of CITI-SENSE, such as volume, velocity, variety, target scenario (2000 events/sec) and the stressed scenario tests.

9.2 Results

In this section the results of all the database solutions for classical CRUD operations are discussed. As the results in these tests output millions of lines, the data is shown with graphs and averages. SenseMark is implemented on MongoDB and the performance results which are compared to the WFS server are discussed in this part.
9.2.1 Write Performance of MongoDB

The SenseMark generates the sensor data events and write them to MongoDB server. Different scenarios are implemented and tested to analyze the write performance of MongoDB. Write performance results of MongoDB are shown in table 9.3.

<table>
<thead>
<tr>
<th>Total no. of sent events/sec</th>
<th>Average writes/sec</th>
<th>No. of events delayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 events/sec</td>
<td>1933-1942 events/sec</td>
<td>58</td>
</tr>
<tr>
<td>5000 events/sec</td>
<td>4685 events/sec</td>
<td>315</td>
</tr>
<tr>
<td>10000 events/sec</td>
<td>8910 events/sec</td>
<td>1090</td>
</tr>
<tr>
<td>20000 events/sec</td>
<td>15750 events/sec</td>
<td>4250</td>
</tr>
<tr>
<td>50000 events/sec</td>
<td>13524 events/sec</td>
<td>36476</td>
</tr>
</tbody>
</table>

Table 9.3: write performance results of MongoDB

The results shown in table 9.3, it has been observed that more events are delayed into the storage if the number of writes increases. As per the requirement for CITI-SENSE that is 2000 events/sec, MongoDB shows reliability for this scenario and 58 events out of 2000 events are delayed that might be due to some background processes or workload on the system. In other words, it could be concluded that MongoDB successfully fulfill the required scenario (2000 events/sec) of CITI-SENSE.

It could be observed that as the total no. of events increases, the no. of delayed events also increase. Hence, the proportionate result shows, more the number of write and more will be the possibility of the delayed events. When stress testing is performed for 50000 events/sec, the MongoDB server crashed and the average writes in that case were only 13524 events/sec. It is expected that higher performance for this can be achieved with a MongoDB database run in a database cluster with multiple CPUs and disks.

9.3 Read Performance of MongoDB as compared to WFS

After generating and storing the data on MongoDB server, the read performance of MongoDB is analyzed and compared on a high level with the CITI-SENSE WFS server on top of a relational databases. Read performance results of MongoDB as compared to WFS service are shown in table 9.4
<table>
<thead>
<tr>
<th>Query operation</th>
<th>Response Time WFS (Average) sec</th>
<th>Response Time MongoDB (Average) sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query all the observations for a specific sensor device</td>
<td>5.86</td>
<td>0.215 sec</td>
</tr>
<tr>
<td>Query all observations for a specific sensor device at a specific point in time</td>
<td>3.47</td>
<td>1.990 sec</td>
</tr>
<tr>
<td>Query all observations for a specific sensor device in a given time period.</td>
<td>4.34</td>
<td>3.57 sec</td>
</tr>
<tr>
<td>Query the latest observation for a specific sensor device</td>
<td>2.89</td>
<td>0.07 sec</td>
</tr>
<tr>
<td>Query the latest observation for a specific sensor device for a specific pollutant.</td>
<td>Time Out</td>
<td>0.003 sec</td>
</tr>
<tr>
<td>Give me all observations for a specific location</td>
<td>Time Out</td>
<td>1.55 sec</td>
</tr>
</tbody>
</table>

Table 9.4: Read performance results of MongoDB

As shown in table, the MongoDB shows better performance as compared to WFS service while calling read operations such type of sensory data events.

It could be observed in the table that while calling a query operation to query all the observations for a specific sensor device, the response time in WFS server is 5.86 sec while in MongoDB the response time is 0.215 sec. In all the required scenarios MongoDB shows results better as compared to WFS service. It is also observed that time out issue is faced in WFS server while calling some read operations whereas MongoDB never show a time out error which proves its reliability and stability.

It should be noted, however, that these results are not directly comparable in terms of system setup. The WFS server was running in an Amazon Cloud server while the MongoDB was running locally on a test machine.

9.4 RDF/Linked Data

Although RDF/Linked Data is considered to be a good approach for sensory data events but when it comes to data events that are continuously sending data to server and increasing the data volume and populating the database, Linked Data is not considered to be a best solution. When this data is mapped to RDF, it gives huge rise to data volume almost 15 times larger as compared to original datasets. There are some performance challenges faced while mapping the CITI-SENSE monthly dumps of tabular data into RDF.

The most important issue is with larger amount of storage data emerging as
a result of RDF mapping. Each monthly dump of CITI-SENSE consists of approximately 1.78Gb of data and when mapped to RDF it becomes almost 12-15 times larger as compared to original CSV data set. In CITI-SENSE use case where we have about 22GB of data for 12 months, when mapped to RDF it would become more than 300Gb and the storage cost will increase a lot. So, it would not be a good solution to map such type of sensor data into RDF which is increasing gradually and giving huge rise to data volume hosted on cloud platform.

Another performance challenge faced while mapping the CITI-SENSE data into RDF is that DataGraft doesn’t support online RDF mapping of CSV data sets whose size is greater than 10Mb. In such case a local RDF mapping could be done using downloadable executable. Moreover, local Java transformation also support CSV datasets up to maximum size of 100Mb and in CITI-SESNE monthly dumps, measurement CSV files size is about 1.5Gb in size and it is not possible to map it directly even with local java transformation. It is required to break CSV dataset into several CSV files of size 100Mb that itself is a challenge and a time-consuming process.

After splitting the larger CSV files into smaller one’s of size 100Mb, when performing transformation and mapping into RDF using downloadable executable, it takes a lot of time to do mapping for each dataset. After mapping 100Mb of CSV into RDF it becomes 1500MB of RDF dataset which is again a challenge to store and publish it on the DataGraft server. Besides this, due to some scalability limit and technical issues it was not possible to move and store 1500Mb of RDF to DataGraft server directly from DataGraft.

Due to these performance challenges and scalability issues, it was decided not to use Graph database for such type of sensor data which is gradually increasing specially in CITI-SENSE use case where there are 2000 sensors each transmitting data event once in a second. In this way, there are 2000 events/sec sent to the server that are increasing the data volume at a very high rate and RDF/Linked data is not suitable solution in such case.

Partially as a result of the SenseMark benchmark the DataGraft solution is now being optimised to handle larger data volumes for ingestion, in order to meet the demands of systems with similar needs to the data load profile of SenseMark. The timing of supporting this is, however, beyond the timeline of this thesis work.
10 Conclusion and Future Work

This chapter highlights the conclusion of this thesis work and also suggests the future work that has a potential for further research.

10.1 Conclusion

The aim of this thesis was to provide a basis for a database benchmark in order to evaluate and benchmark the performance of different database solutions for sensor data management with respect to classical CRUD operations.

The problem statement was addressed by developing SenseMark as a suitable benchmark for the databases to be used for the management of sensor data. It has evaluated WFS as an initial database service and then further evaluated the NoSQL type of database MongoDB (document store) as an alternative database.

This thesis has also conducted a closer evaluation of RDF/Linked Data/Triple stores, where the conclusion was that due to the larger amount of storage data emerging from the RDF conversion this would not provide a higher performance than the initial WFS solution.

The analysis of test results conducted in this project revealed that MongoDB (NoSQL) database is an appropriate solution for sensor data management and also meets the requirements of CITI-SENSE architecture and framework.
10.2 Future Work

The thesis has developed and presented a baseline suitable for future benchmarks of various databases and storage alternatives. With the time and resource limitations of a short master thesis it has not been possible to extensively test and evaluate various alternatives. Future work in progress will be using this work as a basis for more extensive testing and evaluation including cluster database solutions in the Hadoop Big Data Ecosystem.

Future work should focus on evaluation and implementation of SenseMark for some other database solutions for example NoSQL (Cassandra, CouchDB), Graph database solutions (RDF/Linked data), Relational databases (RDBMS) like MySQL or PostgreSQL, Historian databases, Array databases, and Time series databases. Future work should also consider various types of distributed systems and clusters, such as the distribution possibilities available in a Hadoop ecosystem and also the sharding possibilities of MongoDB run in a database cluster.

It is suggested for the future to set the database systems/solutions to be evaluated up in a comparable server/cluster setup. In the current case where the server system is so different with respect to server versus local execution it becomes difficult to evaluate where the time is used – which also might be due to network communication and other issues.

The current version of SenseMark should thus be more viewed as a database benchmark reference model that needs further implementation support to be easily used for the comparison of different database solutions. Future work could also include a generalization of parts of the benchmark code into a common parts for all databases alternatives and into specific parts dealing with the necessary adaptation for one specific database. In the current SenseMark the code has been written specifically only for MongoDB.
11 Appendix

SenseMark Data Events Generator

/*
 * MongoDB
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 */
import java.util.Calendar;
import com.mongodb.BasicDBObject;
import com.mongodb.BulkWriteOperation;
import com.mongodb.BulkWriteResult;
import com.mongodb.Cursor;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodbDBObject;
import com.mongodb.MongoClient;
import com.mongodb.ParallelScanOptions;
import com.sun.xml.internal.ws.commons.xmlutil.Converter;
import java.net.UnknownHostException;
import java.util.List;
import java.util.Set;
import java.util.concurrent.TimeUnit.SECONDS;
/**
 * The tutorial from http://docs.mongodb.org/ecosystem/tutorial/getting-started-with-java-driver/
 */
public class DataGeneration {
  // CHECKSTYLE:OFF
  /**
   * Run this main method to see the output of this quick example.
   *
   * @param args takes no args
   * @throws UnknownHostException if it cannot connect to a MongoDB instance at localhost:27017
   */
  public static void main(final String[] args) throws UnknownHostException {
    // connect to the local database server
  }
MongoClient mongoClient = new MongoClient();

/*
// Authenticate - optional
MongoCredential credential = MongoCredential.createMongoCRCredential(userName, database, password);
MongoClient mongoClient = new MongoClient(new ServerAddress(), Arrays.asList(credential));
*/

// get handle to "mydb"
DB db = mongoClient.getDB("SenseMark_db");

// get a list of the collections in this database and print them out
Set<String> collectionNames = db.getCollectionNames();
for (final String s : collectionNames) {
    System.out.println(s);
}

// get a collection object to work with
DBCollection coll = db.getCollection("SensorDataEvents");

// drop all the data in it
coll.drop();

//Preparing the date
Calendar cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 0);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
cal.set(Calendar.DATE, 1);
cal.set(Calendar.MONTH, 2);
cal.set(Calendar.YEAR, 2016);

//Loop from Date to Date
for (int i = 0; i < 3600; i++) {
    cal.add(Calendar.SECOND, 100); //increment time in seconds for each record
    int value = -1000;
    String property = "";
    if (value %2 == 0){
        property = "CO";
    }else{
        property = "NO";
    }
    for (int l =0; l<100; l++){
        BasicDBObject doc = new BasicDBObject("sensorId", "AT_" + l)
            .append("timestamp", cal.getTime())
            .append("Latitude", 59.78)
            .append("Longitude", 10.86)
            .append("Location", "Oslo")
            .append("info", new BasicDBObject("value", value++)
            .append("measured_property", property));
    }
}
if (value > 5000) { value = -2200; } // reset the value
coll.insert(doc); // inserting the data into MongoDB
}

// get it (since it's the only one in there since we dropped the rest earlier on)
DBObject myDoc = coll.findOne();
System.out.println(myDoc);

// now, lets add lots of little documents to the collection so we can explore queries and cursors
// for (int i = 0; i < 100; i++) {
//    coll.insert(new BasicDBObject().append("i", i));
//}
// System.out.println("total # of documents after inserting 100 small ones (should be 101) " + coll.getCount());

// lets get all the documents in the collection and print them out
DBCursor cursor = coll.find();
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}

// now use a query to get 1 document out
BasicDBObject query = new BasicDBObject("i", 71);
cursor = coll.find(query);

try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}

// $ Operators are represented as strings
query = new BasicDBObject("j", new BasicDBObject("$ne", 3)).append("k", new BasicDBObject("$gt", 10));
cursor = coll.find(query);

try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}
finally {
    cursor.close();
}

// now use a range query to get a larger subset
// find all where i > 50
query = new BasicDBObject("i", new BasicDBObject("$gt", 50));
cursor = coll.find(query);
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}

// range query with multiple constraints
query = new BasicDBObject("i", new BasicDBObject("$gt", 20).append("$lte", 30));
cursor = coll.find(query);
try {
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}

// Count all documents in a collection but take a maximum second to do so
coll.find().maxTime(1, SECONDS).count();

// Bulk operations
BulkWriteOperation builder = coll.initializeOrderedBulkOperation();
builder.insert(new BasicDBObject("_id", 1));
builder.insert(new BasicDBObject("_id", 2));
builder.insert(new BasicDBObject("_id", 3));
builder.find(new BasicDBObject("_id", 1)).updateOne(new BasicDBObject("$set", new
BasicDBObject("x", 2)));
builder.find(new BasicDBObject("_id", 2)).removeOne();
builder.find(new BasicDBObject("_id", 3)).replaceOne(new BasicDBObject("_id", 3).append("x", 4));
BulkWriteResult result = builder.execute();
System.out.println("Ordered bulk write result : " + result);

// Unordered bulk operation - no guarantee of order of operation
builder = coll.initializeUnorderedBulkOperation();
builder.find(new BasicDBObject("_id", 1)).removeOne();
builder.find(new BasicDBObject("_id", 2)).removeOne();
result = builder.execute();
System.out.println("Ordered bulk write result : " + result);

// parallelScan
ParallelScanOptions parallelScanOptions = ParallelScanOptions.builder()
    .numCursors(3)
    .batchSize(300)
    .build();

List<Cursor> cursors = coll.parallelScan(parallelScanOptions);
for (Cursor pCursor: cursors) {
    while (pCursor.hasNext()) {
        System.out.println(pCursor.next());
    }
}

// release resources
//db.dropDatabase();
mongoClient.close();

// CHECKSTYLE:ON
MongoDB Query operations

1. show dbs
2. db.SensorDataEvents.find().pretty()
3. db.SensorDataEvents.find({"sensorId": "AT_10"}).pretty()
4. db.SensorDataEvents.find({ "Location": "Oslo" }).pretty()
5. db.SensorDataEvents.find({ "Location": "Edinburgh" }).pretty()
6. db.SensorDataEvents.find({ "Location": "Oslo", timestamp: {
    $gte:ISODate("2016-02-08T00:00:00Z"), $lt: ISODate("2016-09-20T00:00:00Z") } }).pretty()
7. db.SensorDataEvents.find({ "sensorId": "AQ_10", timestamp:
    { $gte:ISODate("2016-03-01T10:15:00Z")} })
8. db.SensorDataEvents.find({ timestamp: { $gte:ISODate("2016-02-01T11:00:09Z"), $lt: ISODate("2016-02-09T11:10:0Z") } })
9. db.SensorDataEvents.find({ timestamp: { $gte:ISODate("2016-07-01T00:00:00Z"), $lt: ISODate("2016-10-02T00:00:00Z") } }).pretty()
10. db.SensorDataEvents.find({ "sensorId": "AE_48", timestamp: {
    $gte:ISODate("2016-07-01T00:00:00Z"), $lt: ISODate("2016-10-02T00:00:00Z") } }).pretty()
11. db.SensorDataEvents.find({ "sensorId": "AB_49" }).sort({$natural:-1}).limit(1)
12. db.getCollection('SensorDataEvents').find({"sensorId": "AE_10","info.measured_property" : "CO"}).sort({$natural:-1}).limit(1)
12 Bibliography

13. Robinson, D., Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster. 2008: Emereo Pty Ltd. 188.
17. 8 ways the Internet of things will change the way we live and work. THE FUTURE IS SMART, 2016.
19. Mason, R.T., NoSQL Databases and Data Modeling Techniques for a Document-oriented NoSQL Database.


47. CITI-SENSE.