BIGDATA ANALYTICS ARCHITECTURES FOR HVAC ENERGY OPTIMIZATION SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

DOĞAN POYRAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2015

Approval of the thesis:

BIGDATA ANALYTICS ARCHITECTURES FOR HVAC ENERGY OPTIMIZATION SYSTEMS

submitted by **DOĞAN POYRAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Dr. Cevat Şener
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Assoc. Prof. Dr. Tolga Can
Computer Engineering Department, METU ________________

Dr. Cevat Şener
Computer Engineering Department, METU ________________

Assist. Prof. Dr. Adnan Özsoy
Computer Engineering Department, Hacettepe University ________________

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU ________________

Assist. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering Department, METU ________________

Date: ________________

ABSTRACT

BIGDATA ANALYTICS ARCHITECTURES FOR HVAC ENERGY OPTIMIZATION SYSTEMS

Poyraz, Doğan
M.S., Department of Computer Engineering
Supervisor : Dr. Cevat Şener

December 2015, 127 pages

Energy consumption affects both energy bills of the buildings and environment greatly. Especially HVAC systems are the components that consume the most energy in commercial or residential buildings. HVAC stands for heating, ventilating and air conditioning systems in the buildings. In this thesis, data organization, retrieval and processing needs of a HVAC energy optimization system (EOS) have been analyzed, underlying technologies have been examined and architectural solutions have been proposed in order to solve various problems that a HVAC EOS may encounter. This research defines a HVAC EOS in a formal way, so that computer scientists can understand needs of energy domain, specifically HVAC EOS, easier. In addition, this research describes technologies related to BigData and presents architectural examples so that it gives insight about how to solve BigData related problems in energy domain.

In order to build a HVAC EOS, there are several steps throughout the flow of the data. First one is the data stream that is between the sensors in the field and the system. In this thesis, methods to manipulate these data streams are presented so that data can be pre-processed before it is written to a database or a persistent medium. This enables data to be processed much closer to real-time. The second step is continuous processing of the persistent data for forecasting. This operation can be performed in a distributed environment so that when data size is very large, processing power of the several nodes can be used and operation can be completed much faster. The final step is the data visualization. In this step, a user of the system interacts with the HVAC EOS and manually processes some of the data or query data and display it. Again, depending on the data size, this operation can be performed in a distributed environment or data can be stored in a small in-memory medium so that response was returned to the user quickly.

Keywords: BigData, NoSQL, Scalable, Internet of Things, HVAC, Energy Optimization

ÖZ

HVAC ENERJİ OPTİMİZASYON SİSTEMLERİ İÇİN BÜYÜK VERİ ANALİZİ MİMARİLERİ

Poyraz, Doğan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Cevat Şener

Aralık 2015 , 127 sayfa

Enerji tüketiminin hem faturalar hem de çevre üzerinde büyük etkileri vardır. Özellikle HVAC sistemleri ticari ve meskun binalarda en fazla enerji tüketen birimlerdir. HVAC binalardaki ısıtma, havalandırma, ve iklimlendirme sistemlerine verilen addır (İng. Heating, Ventillation, Air Conditioning). Bu tezde HVAC enerji optimizasyon sistemlerinin (EOS) veri yapısı, erişimi ve işlemesi konusunda gereksinimleri incelendi; bu gereksinimlere çözüm oluşturabilecek teknolojiler araştırıldı ve HVAC enerji optimizasyon sistemlerinin karşılaştığı çeşitli problemleri çözmek için sistem mimarileri önerildi. Bu tez bilgisayar bilimi ile uğraşan insanların enerji alanındaki problemleri daha kolay anlaması amacıyla bir HVAC enerji optimizasyon sistemini ve gereksinimlerimi yazılım diline uygun şekilde açıklar. Ek olarak, bu çalışma büyük veri ile ilgili teknolojileri açıklar ve enerji alanındaki büyük veri kaynaklı sorunların çözümü için mimari çözümleri sunar.

Bir HVAC enerji optimizasyon sisteminde veri akışı sırasında birkaç önemli adım bulunur. İlk olarak sensörler ve sistem arasındaki yoğun veri akışı. Bu tezde, veri akışını daha yoldayken işleyebilmeye yönelik teknoloji ve yöntemler anlatılmaktadır. Böylece gelen veriler veritabanına yazılmadan önce gerekli kontrol ve işlemler yapılabilir. Bu şekilde veri gerçek zamana en yakın şekilde işlenebilir. İkinci adım, tahminler için verinin sürekli ve periyodik olarak işlenmesi. Bu işlem, veri boyutu çok büyükse dağıtık bir ortamda yapılabilir. Böylece işleme kümesi içindeki birçok noktanın veri işleme gücü paralel olarak kullanılır ve işlem daha hızlı yapılabilir. Son adım verinin görselleştirilmesi. Bu adımda, sistem kullanıcısı sistemle doğrudan etkileşime girerek elle bazı hesaplamalar yaptırabilir, ya da bir miktar datayı seçip görüntülemek isteyebilir. Veri boyutuna bağlı olarak, bu işlem de dağıtık bir ortamda gerçekleştirilebilir ya da tamamen bellekte çalışan bir veri saklama yöntemi ile kullanıcıya çok daha hızlı tepki verilebilir.

Anahtar Kelimeler: Büyük Veri, NoSQL, Ölçeklenebilir, Nesnelerin İnterneti, HVAC, Enerji Optimizasyonu

To my family and Özlem

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Dr. Cevat Şener for his support and guidance through my research. He motivated, guided and communicated very well with me. I am glad to have worked with him.

I also want to thank Şahin Çağlayan, CTO of Reengen Energy, for his support and guidance about the domain that I was not familiar with. During my work, he always helped me to understand the domain and answered my questions clearly so that I was able to produce solutions to the problems in the energy domain. In addition, he provided me data and information for my tests so that my research is about a real system, not a fictional one.

In this research, Amazon Inc. supported me financially so that I was able to deploy my solutions in their cloud platform and perform tests. Therefore, I also thank to them for their moral and material support.

I also want to thank to my family for their support. They helped me with my non-educational stuff so much and they unburdened me from a lot of miscellaneous work.

Lastly, I want thank Özlem Özdoğan separately. She has always supported, encouraged and helped me whenever and wherever I need. I wish she does the same for the rest of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IoT	Internet of Things
RDBMS	Relational Database Management System
RFID	Radio-Frequency Identification
NoSQL	Not only SQL
ACID	Atomicity, Consistency, Isolation, Durability
CAP	Consistency, Availability, Partition Tolerance
PODC	Principles of Distributed Computing
BASE	Basically Available, Soft state, Eventually consistent
SQL	Structured Query Language
SSD	Solid-State Disk
JSON	JavaScript Object Notation
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language
BSON	Binary JSON
HTTP	Hyper-Text Transfer Protocol
API	Application Programming Interface
HDFS	Hadoop Distributed File System
EMR	Elastic Map-Reduce
CQL	Cassandra Query Language
Hive-QL	Hive Query Language
IOTMDB	Massive IoT Data Based
SG	Smart Grid
CDR	Cloud-based Demand-Response
HVAC	Heating, Ventilating and Air Conditioning
EOS	Energy Optimization System
BMS	Building Management System
AHU	Air Handling Unit
AWS	Amazon Web Services

EC2	Elastic Compute Cloud
JVM	Java Virtual Machine
IP	Internet Protocol
JDBC	Java Database Connectivity
RDD	Resilient Distributed Dataset

CHAPTER 1

INTRODUCTION

Internet of Things (IoT), sometimes referred as internet of everything, refers to networked interconnection of everyday objects, which are often equipped with ability of generating data [41]. IoT concept increases interaction of networked devices with different devices and human beings. Thanks to rapid advances in underlying technologies, IoT technology is spreading all over the world, in almost every aspect of our lives. IoT allows us to collect data remotely from many devices deployed in the field and control them through existing networking infrastructures.

Nodes in the IoT consist of a wide variety of devices. For example, automobiles with built-in sensors, meteorological sensors, sensors of an army vehicle, security cameras and so on. All of these devices collect data from the field and send them to different servers. Considering all the data collecting devices all over the world, enormous amount of data flows around the world.

According to Cisco, global mobile data traffic will reach 15.9 Exabyte per month or 190 Exabyte annually, increasing nearly 11-fold from 2013 to 2018 [21]. Handling this amount of data is not possible with traditional storage and processing technologies. In addition, according to National Cable & Telecommunications Association (NCTA), the number of active mobile devices around the world will be increasing exponentially in the following years as can be seen in Figure 1.1 [16].

Energy domain is one of the largest subsets of IoT network. An IoT application in energy domain is HVAC EOS. HVAC stands for heating, ventilating and air conditioning. It is the technology for maintaining indoor air quality by keeping the temperature

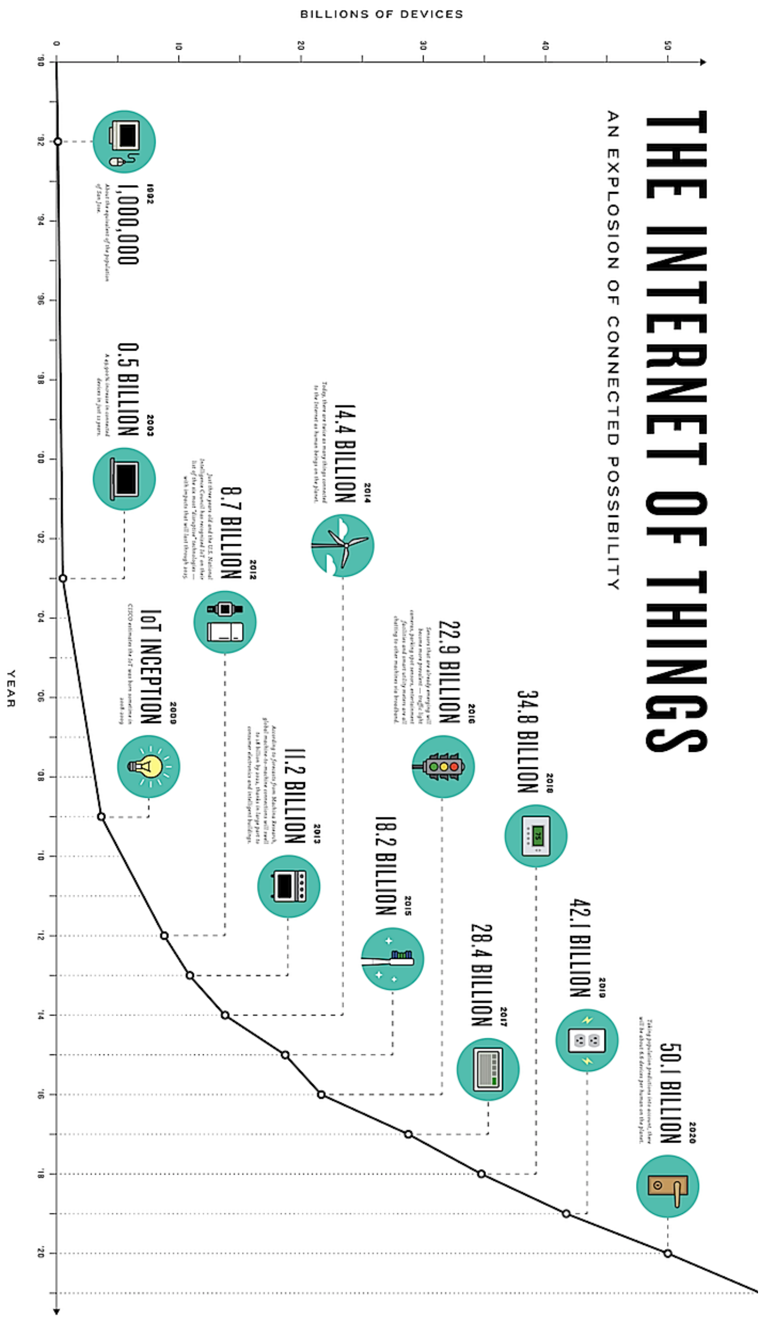
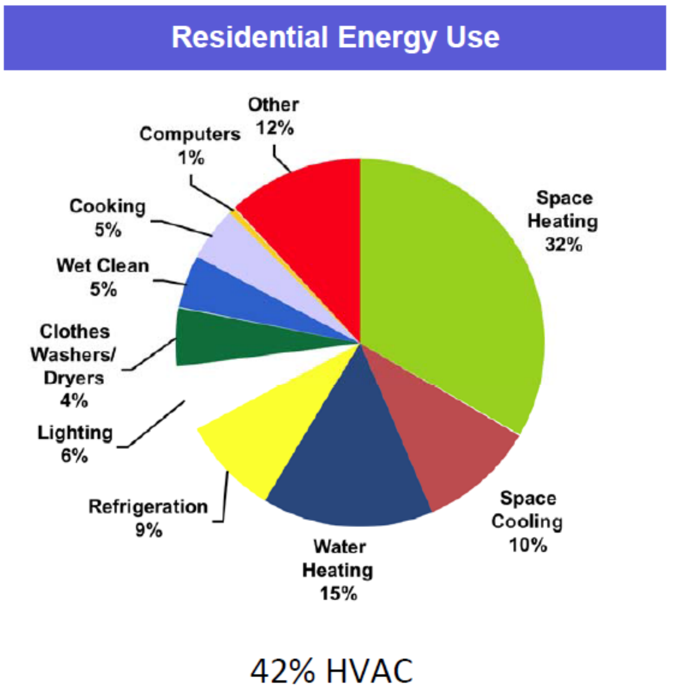


Figure 1.1: Increase in Number of Active Mobile Devices [16]

at a desired level and air clean by replacing air in any indoor space. According to U.S. Department of Energy, HVAC systems consume the most energy in both residential and commercial buildings [34] as can be seen on Figure 1.2. Therefore, by optimizing the energy consumption of HVAC systems enormous amount of energy can be saved. At this point, HVAC EOS comes into action. These systems are built on top of an existing HVAC system. In order to sense and control the status of an HVAC system, HVAC EOS uses several sensors and remote controllers, which create huge amount of data flow that we can classify as BigData.



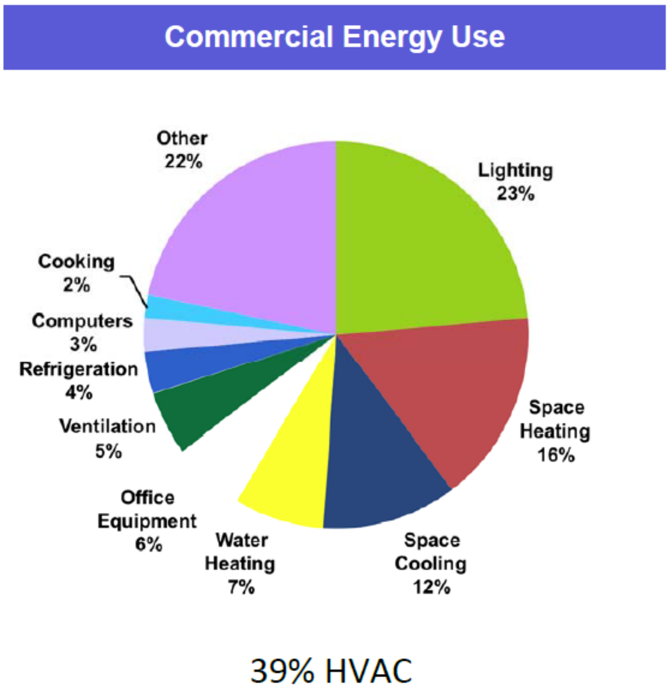


Figure 1.2: Energy Consumption of HVAC Systems [34]

When data volume and flow rate is not very high, data can be handled easily with old methods like RDBMSes. However, as these parameters increase, handling the data gets harder and more expensive in terms of time and money. For an HVAC EOS, several sensors are deployed in a floor in order to just collect data. If the system is set up all around a city with one million population, it makes more than a few millions of sensors. In addition to data collection, there will also be nodes for controlling the HVAC equipment remotely. Considering this amount of data generating components, only the data collection rate and data amount will exceed performance limits of traditional methods.

In order to implement and set up HVAC EOS in large scale, these problems need to be solved properly. Moreover, as the energy needs of the world increase, optimizing HVAC systems efficiently will save lots of energy and resources. Since HVAC EOS operate on real-time data, they have strict operational and performance requirements. In order to meet those requirements, problems caused by data volume and rate need to be solved. Here are some of the challenges that are caused by huge data amount and rate:

- In an HVAC EOS, there are several types of data-generating devices and each device generates data in different format even if devices generate data for same agent. Therefore, organizing this kind of data in a single table is not possible with relational databases.
- Storing huge data volumes in RDBMSs cannot meet performance needs.
- Even if data are stored efficiently somehow; analyzing, processing and visualizing this huge data efficiently are other important problems.

We decided to focus on these problems and conduct this research because there was no clear solution for a complete HVAC EOS in the literature. Contributions of this thesis are in a few different topics. First, this thesis describes relation of HVAC EOS and BigData technologies. Then, it defines data organization, retrieval and processing needs, operational and performance requirements of an HVAC EOS system formally. Finally, it proposes a final software architecture. In order to do these contributions, first this thesis defines BigData and NoSQL technologies. Then, describes how they are related to the energy domain and HVAC EOS. After that, details of HVAC EOS is expressed in a formal way. Tools and technologies that are used for BigData solutions are analyzed in detail and some popular tools and frameworks are examined and compared. These include NoSQL databases and distributed data processing frameworks. Rather than focusing on high-level cloud structure or low-level architectural solutions, we proposed several software architectures for different use-cases. After these proposals, test results with different tools are presented. Depending on the results, characteristics of tools are discussed. Moreover, which tool is better for what kind of purposes is explained. Consequently, analysis of proposed architectures and tests are discussed in detail and an optimal software architecture is proposed, considering the requirements of the HVAC EOS.

Rest of the thesis is organized as follows: In Chapter 2, BigData and NoSQL are defined. Then, technologies related to them are described with technical details. In Chapter 3, previous researches related to this topic are presented. We analyzed what previous studies achieved and what is missing in them. Then, in Chapter 4, a reference HVAC EOS is described in detail. After presenting technologies and the system, in Chapter 5, several software architectures were proposed for different needs of the described HVAC EOS. Then, in Chapter 6, first the testing environments are described. Next, several tests are performed with the tools used in 5. After the tests, results are presented and they are discussed in detail. After discussing the results, according to the results obtained, we illustrated an optimal software architecture. Finally, the thesis is summarized and remaining works for the future are given in the conclusion chapter (Chapter 7).

CHAPTER 2

BIGDATA FOR HVAC ENERGY OPTIMIZATION SYSTEMS

BigData is a term used for very large data sets. These sets are so large or complex that traditional methods for storing and processing data are either inadequate or very inefficient. All over the world, data sets are growing rapidly because the equipment that generates digital data gets cheaper and data sources get widespread. These data sources include information-sensing mobile devices, software logs, cameras, microphones, RFID readers and wireless sensor networks [37].

Traditional RDBMSs and data processing and handling mechanisms often have difficulty in handling BigData. According to Adam Jacobs, the processing such huge data sets requires "massively parallel software running on tens, hundreds, or even thousands of servers" [28].

How much data are considered “BigData”? Answer of this question depends on who is and answering and when. Currently, if a data set is over a few terabytes; it is classified as big data [38]. The size limit for big data is where a data set is so large that it extends multiple storage units. Another parameter for the size limit is where traditional RDBMS technologies begin to have a hard time handling the data set. These parameters depend on the technology of physical devices, of course. As technology develops and devices get stronger, the limit gets higher but still there is a limit. At some point, horizontal scaling is needed for better performance. BigData is often described with its several Vs, which define characteristics of a data set. Here are some of the important Vs that define BigData:

Volume, refers to the size of the gathered data. Considering the whole Web, enormous amount of data are generated and shared every day. Just on Facebook, billions of new messages are sent and millions of new pictures are uploaded every day. Storing and processing this much data are not possible with traditional data storage and processing techniques. The big data technology enables us to store and use this data with severing different techniques of data storage and retrieval. For example, it stores parts of the data at different physical locations and combines them with software when needed.

Velocity, means pace of generation of new data and movement of existing data between nodes. Thinking of the messages in Facebook, which is mentioned above, we can have an idea how much data movement occurs.

Veracity, refers to accuracy of the data. While the gathered data amount increases, lots of useless, corrupted or abnormal data are also generated. Thanks to the new BigData technologies, we are able to handle and clean these unnecessary data.

Variety, refers to different sources and types of data generated. In traditional systems, format of most of the data are predefined and fits into rectangular tables. All rows of a table are in same structure and have same columns. However, most of the world's data are becoming unstructured like photos, messages, conversations, data from different sensors etc. Therefore, data does not fit into predefined rectangular tables. A new column may arise anytime. Big data technology helps us to store and retrieve various data together.

Volatility, means for how long a data should be considered as valid and how long it should be stored. Purging the unnecessary data will relieve the system as it may save a lot of space.

Value, means business value of the data, not numeric value. Technology allows us to cope with all other Vs of big data, however if we cannot use the data to interpret something out of it or assign a meaning to it, those data are not good for anything. Big data technologies are useful as long as we can extract a value out of it, else those data are useless.

Variability, means inconsistencies of value or meaning of data at different times or in different concepts. For example, if one is performing language processing, a word may define different things in different sentences. Or a numeric value of a system may be classified 'very low' in the morning and 'very high' in the evening. Therefore, in situations like this, data need to be classified depending on its meaning, rather than its value.

Visualization, refers to visual representation of the collected big data or results extracted from it. Visualization is one of the key parts of decision making. Without a good visualization model, it would be very hard to interpret meaning of the flowing data. Moreover, big data can be accessible only be visualizing it properly. Raw big data cannot be read and understood.

Before moving to big data solutions, some concepts about traditional RDBMS and NoSQL design needs to be clarified.

2.1 ACID Properties in Relational Databases

In computer science, ACID is an abbreviation used for a list of properties that assure reliable processing of database transactions. These properties are present in most of the traditional RDBMSs. Andreas Reuter and Theo Härder, called this property set ACID in 1983 [27]. They defined these properties as following:

Atomicity: Each transaction has to be atomic. It means if a transaction is successful, all parts of it is completed successfully; if any part of it fails, the entire transaction fails and database content and state is left unmodified. An atomic database must assure in every situation including software failures, hardware failures, energy shortages etc. parts of a committed transaction is inseparable.

Consistency: At the end of each completed transaction, either successful or unsuccessful, database is left in a valid state. Any operation in the database must be valid according to all of the defined rules. These rules include constraints, cascades, triggers, and any combination of these. This property does not assure transaction validity. However, it assures that no application-level programming errors violate defined database rules. Consistency property also ensures that any changes to values in an instance are consistent with changes to other values in the same instance. There are four consistency levels [26]:

- · Degree 0 - a transaction does not overwrite data updated by another user or process ("dirty data") of other transactions.
- · Degree 1 - degree 0 plus a transaction does not commit any write until it completes all its writes (until the end of transaction).
- · Degree 2 - degree 1 plus a transaction does not read dirty data from other transactions.
- · Degree 3 - degree 2 plus other transactions do not read dirty data before the transaction commits.

Isolation: Any operation within a transaction is hidden from other transactions that are running concurrently. Isolation is ensured by use of synchronization methods. Isolation property ensures that both concurrent and serial execution of transactions results in the same system state. Depending on the implementation, effects of an unsuccessful transaction might be invisible to other concurrent transactions. The four consistency levels also define isolation level between transactions.

Durability: Once a transaction is completed and its results are committed to the database, the system guarantees that results of the transaction survive any malfunctions and they are stored permanently. In order to survive through hardware failures and power loss, transaction results must be stored in a persistent memory.

2.2 CAP Theorem

CAP theorem, also known as Brewer's theorem, is first presented as a conjecture by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). In 2002, a formal proof was published and the conjecture was rendered as a theorem by Seth Gilbert and Nancy Lynch [25]. However, it is not completely accepted as a theorem and argued.

Theorem states that a distributed system has three desirable properties, which are:

Consistency, means a read sees all previously completed writes and all nodes see the same data at the same time.

Availability, means every read and write request receives a response about result of the request, either a success response or failure response.

Partition tolerance, means system continues to provide guaranteed services and properties even if a partial network failure occurs and prevents some machines from communicating each other or cause some messages to be lost on the way.

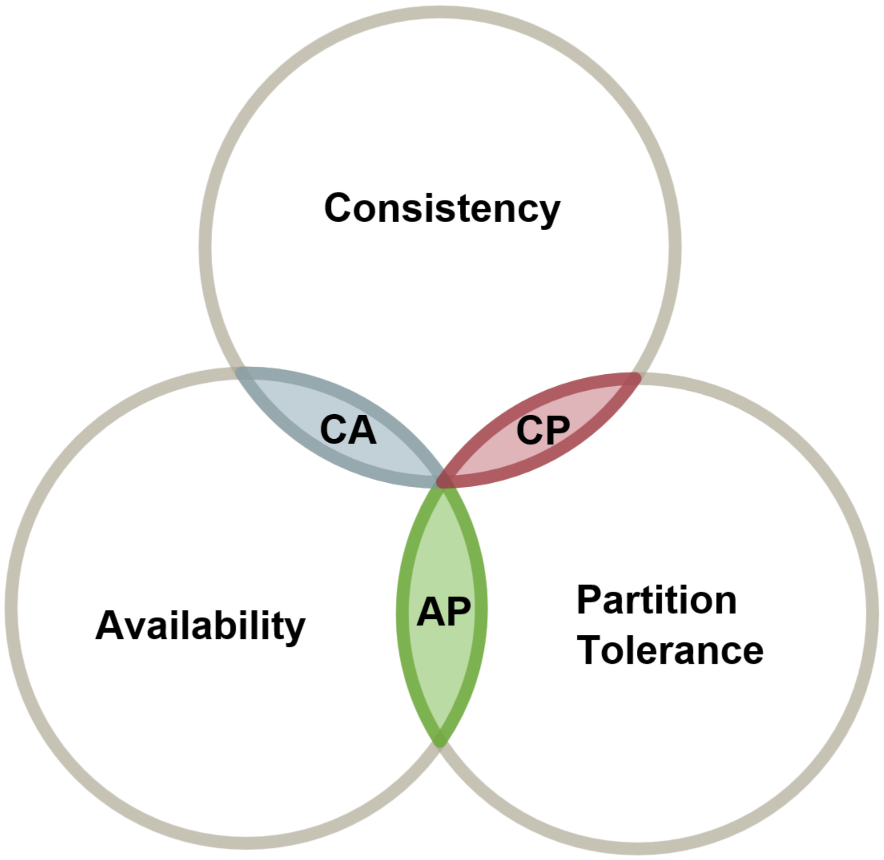


Figure 2.1: Graphical representation of CAP Theorem

The theorem states that a distributed computer system cannot provide all of the properties above simultaneously. This impossibility enforces choice of two out of three CAP properties and leaves three viable design options: Consistency-Availability, Consistency-Partition Tolerance and Availability-Partition Tolerance. Further consideration shows Consistency-Availability is not a logical option. Because, if a system is not partition-tolerant, it will be forced to give up Consistency or Availability at some point [25]. Therefore, a more modern interpretation of the theorem is: during a network partition, a distributed system must choose either Consistency or Availability [9]. Graphical representation of CAP theorem is shown in 2.1.

2.3 BASE Properties

BASE is an alternative property set and database model to ACID. Characteristic property of BASE systems is they provide eventual consistency. BASE model is embraced by NoSQL databases. BASE can be described as opposite of ACID. While ACID forces consistency for every operation, BASE accepts database consistency to be in a soft state. BASE is an abbreviation of following properties:

Basic Availability: This property states that service will be available even in the presence of multiple failures among the system. There will be a response to any request. That response could be failure to retrieve data but service is still responds. Even if a malfunction disrupts access to some part of the data, it will not cause a complete system failure.

Soft State: State of the system could change over time, even if there is no transaction at the moment. System might be sending messages to nodes provide guaranteed properties, such as eventual consistency.

Eventual Consistency: The system will eventually be in a consistent state once it stops receiving input. Received data will propagate through whole system at some point. However, there is no guarantee about when system will be consistent, because it continues to receive input and does not check overall consistency before moving on to processing the next input.

2.4 NoSQL for BigData

The word NoSQL is combination of two words: No and SQL. The term is usually interpreted as “Not only SQL”. Some have proposed the term NonRel as an alternative.

Basically, the term NoSQL implies that it is a technology that counters standard SQL or traditional RDBMS logic. NoSQL refers to a class of products and technologies about storage and manipulation of data, rather than a single product or technology.

Traditional RDBMSs have very solid place in most organizations. They meet the needs of legacy applications and are enough to store and process organization-wise data. However, as the data volume increases, mostly for massive web-scale and IoT data, there emerge problems for RDBMSs. Companies begin to look for alternatives to legacy systems in order to increase the performance beyond capabilities of existing systems or implement a cheaper solution. At this point, a new concept called "BigData" rises and its solutions come in handy.

As the use of the Internet gets widespread, it began to be used in every area. Consequently, overall data size flowing around the internet began to increase exponentially. Here are some problems of huge data sets that is hard for RDBMSs to handle:

- An RDBMS stores relational data. Handling relationships of multiple massive-sized tables is expensive.
- RDBMSs have predefined tables for data; however in the modern Internet and IoT, there are very different types of data. If we create a table for each data schema, we will need to handle lots of tables and their relations.
- An RDBMS sticks to ACID properties, namely meets only 'Consistency' property of CAP theorem. RDBMSs are available as long as it can handle the traffic but almost never susceptible to 'Partition Tolerance'. This means an RDBMS is only vertically scalable.
- When hierarchical storage is needed, an RDBMS cannot handle that kind of data. Developer has to link them using relations among tables or rows.
- RDBMS data are hold and operations are executed in hard drive, which throttles the performance when there are multiple operations occurring concurrently.

NoSQL technology aims to solve these problems, but also has a price of course. Here are some situations where RDBMS is a better than a NoSQL database:

- RDBMS technology has a long history. Therefore, it has greater community and supported by greatest companies like Microsoft, IBM or Oracle. So RDBMSs have better customer support while NoSQL systems are mostly supported by start-ups except a few exceptions.
- Most NoSQL data stores do not support an SQL-like language. Even if they have, their functionality is not anywhere near traditional SQL. So RDBMS is still better for query coding in query intensive environments.
- If we need to keep relations among different tables and query accordingly, it is very easy to do with traditional SQL with join operations. Most of the NoSQL data stores do not support join operations. Therefore, common data have to be duplicated in all relevant rows.
- In high transaction based applications, RDBMSs are still better choice since they focus and guarantee atomicity, consistency and integrity of the data.
- If any of the ACID properties are critical for the application, RDBMS is better choice.

NoSQL databases aim to solve big data-related problems by providing different storage and data retrieval mechanisms than traditional RDBMSs. Some of the mechanism differences implemented by NoSQL databases:

- They do not use relational data model. They are specialized to store unstructured data while RDBMSs store structured data.
- By the nature of their design, NoSQL databases works very well on distributed clusters. Moreover, adding and removing nodes to the system is very simple. Eventually, more performance can be gained by adding nodes to the system. Nodes can be cheap ones. With multiple cheap nodes, one can gain performance more than a very expensive single node RDBMS, with much less total cost.
- Data amount stored on an RDBMS is mainly depends on physical memory of the system. On the other hand, NoSQL databases do not have such limits as

they can be scaled horizontally. Total storage limit is total storage of nodes in the cluster, and it can be increased and decreased easily.

- RDBMSs use fixed schema and data model. NoSQL databases have different categories based on their data models. For example; key-value store, document store, column-based data store. Appropriate one can be chosen for an application.

- Most NoSQL databases have caching facility integrated. By caching data in system memory, they boost input/output performance.

- NoSQL databases favor partition tolerance over consistency and atomicity. They mostly stick to BASE model. Partition tolerance is not an option; it is present by their nature. However, level of consistency and availability may change among different products.

2.4.1 NoSQL Database Types

NoSQL databases are categorized depending on their data model. Here are the most common ones:

2.4.1.1 Key-Value Stores:

Databases of this type use a map or dictionary-like data model. This is the simplest model among others. Each data record is stored with a corresponding key that points to actual value. Therefore, a key can exist at most once.

Key-value stores have different types depending on data management properties. Some of them support storing the data in memory. Some others support ordering of keys, allowing efficient query of a range of keys.

Key-value stores usually provide three basic operations:

- Get(key), returns value pointed by the key.

- Put(key, value), inserts the key-value pair as a new record or overrides if key exists.
- Delete(key), removes the entry associated with the key.

The most used examples of this category are Riak, Redis and DynamoDB:

Riak is a distributed NoSQL key-value store, written in Erlang. It provides mechanisms for high availability, fault tolerance, operational simplicity, and scalability [18]. It has both open source, enterprise and cloud storage version, which are supported by Basho Technologies. It implements principles and techniques from Amazon's Dynamo paper. Riak is licensed with a freemium model.

Redis is an open source key-value store, which is currently sponsored by Pivotal Software. It has bindings and libraries for lots of languages. Redis maps keys to different types of values [17]. It supports abstract data types other than just strings. For example lists of strings, sets of strings, hash tables with string keys and string values. Redis holds the whole data in memory. In previous versions, it has feature to store some of the data on the disk, however this feature is deprecated. It supports master-slave type of replication. A master node replicates the data to several slave nodes. A slave node may be master to another slave. Redis runs as a single threaded single process. Additionally, it provides expiration mechanisms for keys.

DynamoDB is a fully managed key-value store service that is offered by Amazon.com [13]. DynamoDB allows users to purchase throughput, rather than storage like other Amazon.com services. DynamoDB offers Hadoop integration via Elastic MapReduce, which is another Amazon.com service. DynamoDB can store and retrieve any amount of data and serve any level of traffic. It spreads the data and the traffic over sufficient number of servers to handle requested throughput and data amount. All data are stored on solid state disks (SSDs) and automatically replicated across multiple physical locations. There is almost no maintenance cost for DynamoDB, just throughput cost. When user creates a table and specifies throughput, there is nothing else to do. Users do not have to watch hardware, do the setup and configuration or worry for cluster scaling. There is no storage limit. DynamoDB scales itself according to the data amount. For partitioning, consistent hashing is used. There is no master node in DynamoDB. Replica synchronization is decentralized. Provides both strong and eventual consistency. It is configurable for each query. Additional primary keys can be defined. When retrieving data by primary key, retrieval is very fast and speed almost always constant. When transferring data from DynamoDB to a client, data are transferred in JSON form. Therefore, total number of transferred bytes for a query is size of JSON representation of the object returned, which may be disadvantage in some situations. Because, as number of items in a response increases, the transfer time also increases directly proportional. It allows two types of primary keys to be defined: hash type primary key and hash and range type primary key.

Hash type primary key: The primary key consists of only one attribute (column). An unordered hash index is built on the primary key attribute. A unique hash key identifies each item. Only get operation is supported while using this primary key.

Hash and range type primary key: The primary key consists of two attributes, one is the hash attribute and the other is the range attribute. An unordered index is built on the hash key and a sorted index is built on the range key. An item is identified by these two attributes together. While using this primary key, user can issue range queries based on range index.

DynamoDB supports secondary indexes as well. In addition to get and query operations, there is another operation supported, scan operation. In scan operation, each item in the tables is visited and if it satisfies the provided condition, it is returned.

2.4.1.2 Document Databases:

Documents databases stores data as ’document’s. A document may contain many different key-value pairs, key-array pairs or nested documents. Data are encoded in some standard format or encoding like XML, YAML, JSON or BSON. Contents of the documents do not have to be structured in the same way.

Documents in the database are accessed via a unique key that is associated with the document. A document can be queried with this unique key like a simple key-document lookup or based on their contents. Document databases keep attribute meta-data to allow querying based on document contents.

The most common examples of this category are MongoDB and Apache CouchDB:

MongoDB is a document-oriented data store. It holds data in BSON format, which is similar to JSON in dynamic schemas. MongoDB is free and open source [15]. It supports search by specific fields, range queries and regex searches. Queries can return specific fields rather than whole document. Any field can be indexed manually. It uses replica sets as replication strategy. Primary and secondary replicas store the exactly same copy of the data. By default, primary replica performs all reads and writes. When a primary replica fails, one of the secondary replicas is chosen as primary. Secondary replicas can perform read operation, however in this situation data will be eventually consistent. JavaScript functions can be used in queries. They are executed on database.

CouchDB is an open source document-based data store. It focuses on ease of use. It stores data in JSON format, uses JavaScript as query language and MapReduce and HTTP as API [3]. It implements 'Multi-Version Concurrency Control', meaning conflicts are left to application to resolve. It provides document-level ACID properties with eventual consistency.

2.4.1.3 Wide-Column Stores (Column-Based Databases):

These databases are optimized to store and process very large data sets distributed over many machines. Column stores store the columns of data together, rather than rows. This allows very quick access to any specific column in a row.

At higher level, each table holds key-row pairs and each row contains key-value pairs. However, a key of a row can consist of multiple columns. Since the columns are stored together, when only specific columns are needed, not all columns have to be transferred. This method allows very fast access even if a row contains hundreds or thousands of rows.

Databases of this type are very useful for data sets that have very large and varying number of columns. The most common examples of this category are Apache HBase and Cassandra:

HBase is an open source data store written in Java. It implements the principles in Google's BigTable. It runs on HDFS and provides BigTable-like capabilities for Hadoop [5]. The tables of HBase can be supplied to MapReduce tasks of Hadoop as input or output. Row operations are atomic, with row-level locking and transactions. Data are partitioned by range. Multiple versions of data are stored with timestamps, in order to resolve conflicts. Queries can be performed by key lookups or MapReduce jobs. In CAP theorem, HBase covers Consistency and Partition Tolerance properties.

Cassandra is an open source column-based data store. It can handle very large datasets, provides high availability with no single point of failure and asynchronous masterless replication [2]. Consistency level can be modified for each query. Primary key can consist of multiple columns. The first column of the primary key is used as partition key. Data are distributed among the cluster according to the partition key. Rest of the columns of the primary key is used as clustering key. In a partition, data are sorted and stored according to clustering keys. Any other column can be indexed separately from the primary key. Every node in the cluster has the same role. Replication strategy and factor is configurable. MapReduce is supported with Hadoop and Spark integrations. Apache Pig and Apache Hive are also supported. Cassandra provides an SQL-like query language called CQL (Cassandra Query Language). There is no locking mechanism in Cassandra. The replicas are updated asynchronously. It supports supercolumn concept, which is another level of column grouping. For partitioning, consistent hashing is used. Any operation on a row is 'atomic per replica'. Depending on the load of servers in the cluster, nodes are moved on the consistent hash ring in order to balance processing load of the nodes. Data are persisted to local files instead of a distributed file system.

2.4.1.4 Graph Stores:

Instead of a data model consisting of tables, rows and columns, a flexible graph model is used. Data can be easily transformed by modifying connections in the graph.

- Graph stores use edges and nodes to represent and store the data.
- Nodes are organized according to their relationships with other nodes. The relationships are represented by edges between nodes.
- Nodes and edges have defined properties.

Databases of this type are useful for storing graph like data. For example, network topology, social relations and road maps.

The most common example of this category is Neo4J. This category is not suitable for our case and operational scenarios. Therefore, this category is out the scope of this research and there will not be detailed information, tests or evaluation about graph stores.

2.4.2 Comparison of Selected Databases

During tests of this research, DynamoDB and Cassandra will be used for tests as delegates of their categories. DynamoDB is chosen for its ease of deployment and management. In addition, DynamoDB has direct and easy integration with other Amazon Web Services products like Amazon EMR (Elastic Map-Reduce) and Apache Hive. Cassandra is chosen between two most popular column-based databases: HBase and Cassandra. Cassandra is preferred because it favors 'Availability' and 'Partition Tolerance' while HBase favors 'Consistency' and 'Partition Tolerance'. Moreover, HBase has row-level locking for synchronization, which is an unwanted feature for our case. In the example system described in 4, consistency is not as important as availability since there is not any concurrent update operations on a single row, only concurrent reads may occur and this does not affect data consistency. This section will compare selected databases' consistency and query possibilities provided to the user.

2.4.2.1 Consistency

Both of them offer configurable consistency levels. DynamoDB has two consistency modes: eventual and strong consistency. In eventual consistency mode, the read data may not be up to date or exist on all of the replicas. Therefore, user may receive out of date data. In strong consistency mode, system waits for an answer from all of the replicas before returning to the user. This guarantees that if a data record is written to any of the replicas, that data will be retrieved. Cassandra has more consistency levels that can be configured. These are:

ONE: Data must be written to or read from at least one replica node.

TWO: Data must be written to or read from at least two replica nodes.

THREE: Data must be written to or read from at least three replica nodes.

ALL: Data must be written to or read from all replica nodes in the cluster for that partition.

EACH_QUORUM: Data must be written to or read from a quorum of replica nodes in all data centers.

QUORUM: Data must be written to or read from a quorum of replica nodes.

LOCAL_QUORUM: Data must be written to or read from a quorum of replica nodes in the same data center as the coordinator node. This mode avoids the latency of inter-data center communication.

LOCAL_ONE: Data must be written to or read from at least one replica node in the local data center.

ANY: This mode is available for write requests only. A write must be written to at least one node. Even if all of the nodes for the given partition key are down, write can still succeed after a hinted handoff has been written. Hinted handoff is a mechanism for detecting nodes that are down and retrying the write request.

For both databases, these consistency levels can be configured for each request.

2.4.2.2 Query Types

Dynamo supports two types of primary keys. Hash key or hash and range key together:

- *Hash Type Primary Key:* In this case the primary key is made of one attribute, a hash attribute. DynamoDB builds an unordered hash index on this primary key attribute.

- *Hash and Range Type Primary Key:* In this case, the primary key is made of two attributes. The first attribute is the hash attribute and the second one is the range attribute. DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute.

Since DynamoDB is a key-value store, supported operations are relatively simple. It supports three types of operations for data retrieval: Get, Query and Scan. The Query operation enables user to query a table using the hash attribute and a range filter. If the table has a secondary index, user can also Query the index using its key. One can query only tables whose primary key is of hash-and-range type; also any secondary index on such tables can be queried. If primary key is only hash type, only rows of specific keys can be retrieved rather than querying multiple rows. This operation is called Get. Query and Get is the most efficient way to retrieve items from a table or a secondary index. DynamoDB also supports a Scan operation. This operation reads every item in the table or secondary index and returns ones that satisfy the given conditions. For large tables and secondary indexes, a Scan can consume a large amount of resources. Scan operation is useful when retrieving very large amount of items from the table or retrieving data by non-indexed attributes.

Cassandra uses an SQL-like language: CQL (Cassandra Query Language). Insert queries are almost same as SQL except for a few syntactic differences. The different part is select operation. Select supports basic operations of select in SQL like specifying columns to select, retrieving count, etc. However, it does not support join operations.

Cassandra supports compound primary keys. First column of the compound key is used as partition key and others are used as clustering key. Node of the data is decided by partition key and ordering is performed by clustering keys. In order to query for a compound key, user does not have to provide all of the key columns. However, in order to query for a column, user has to provide key columns, which are before that column in the primary key definition.

Partition key column supports only equality operation. In Cassandra, partitioner classes are responsible for deciding what data is placed on which nodes in the cluster. It is supported only if partitioner is an ordered one, but that type of partitioners are not good for partitioning data equally over the cluster. Clustering key columns supports =, <, >, <= and >= operations. 'IN' operation is supported wherever equality operator is supported.

Cassandra has an "ALLOW FILTERING" operator. This operator is used to force Cassandra to execute a query that may be expensive. These queries are only can be executed by retrieving all of the rows and then scan them for the given condition. For example if someone tries to query only for a range of values of a clustering column (usually >, <, <= and >= operators) without providing a complete key, Cassandra cannot retrieve them by their unique key. It has to retrieve all relevant rows and scan them. If user knows what he/she is doing, 'ALLOW FILTERING' clause forces Cassandra to execute the query.

2.4.3 Apache Hive for BigData Platforms

Data warehouse is a central system to integrate and access data from one or more separate sources. Apache Hive is a data warehouse software designed for storing and accessing data in BigData platforms. It enables querying and managing large data sets residing in distributed storage [6]. It runs on a Hadoop cluster and it provides following features:

· Mechanisms to work with different data sources.

· Access to files stored directly in HDFS.

· Query execution via MapReduce.

Hive has a simple SQL-like query language called Hive-QL. This language also allows programmers to plug in their custom mappers and reducers to perform complex analyzes, that is not supported by built-in capabilities, on data. Hive can store data either itself or define external tables to connect to other NoSQL data stores like DynamoDB or Cassandra. If the connected database does not support an SQL-like language or user is not familiar with it, Hive provides an easy language to user in order to access data.

Hive is suitable in following situations:

· Data access operations which are not possible or takes very long time with traditional methods.

· User wants to execute the query with map-reduce paradigm.

· User wants to implement custom mapper and reducer and process the data while executing the query.

Since Hive runs on a Hadoop cluster, its operations include two-stage disk operations and a lot of data movement is needed since data are distributed over HDFS. These characteristics bring a lot of overhead. Therefore, if data amount is small, executing it on Hive might bring large overhead instead of performance gain. Therefore, Hive is suitable for processing huge data sets in background. For user interactions and operations that need rapid response, it is better to use some other method.

2.4.4 BigData Processing

As described before, storing and processing BigData are very hard with traditional technologies. In order to store big data, NoSQL technologies was defined before. In this section, technologies for processing big data will be described.

2.4.4.1 Batch Processing with Map-Reduce

MapReduce is a programming model designed for processing and generating large data sets with a parallel and distributed algorithm that runs on a cluster [22]. Map and reduce functions are commonly used in functional programming. Although their purpose is not the same as MapReduce, they inspired the MapReduce model. In MapReduce, the main improvement of these functions is scalability and fault-tolerance provided for applications that work on BigData by optimizing the execution engine. In other words, this model will not result in faster execution if it is used in a single-threaded environment or application.

MapReduce framework is composed of two main procedures that needs to be implemented by the application: Map and Reduce procedures. Map procedure performs initial filtering and sorting on key-value pairs that are provided as an input to the system. Then, it sends it to associated reducer. Reduce procedure takes output of map operation as input. It performs summary operation by processing the provided input. MapReduce operation can be defined in three steps: Map, Shuffle and Reduce. With intermediate details, we can describe the whole operation in seven steps [22]:

1. The MapReduce framework first splits the input files into M pieces (typically 16 MB to 64 MB, configurable via an optional parameter). Then, it starts copies of the program distributed over the whole cluster.

2. One of the copies of the program is master. It assigns work to the other workers. Between M map tasks and R reduce tasks, master picks idle workers and assigns them a task to execute.

3. When a worker is assigned a map task, it reads the content of the input split assigned to that map task. It parses key-value pairs from the input data and passes each pair to map function that is implemented by the user. Then, intermediate key-value pairs are produced by the map function and they are buffered in the memory.

4. Buffered intermediate pairs are written to the local disk periodically. They are partitioned into R regions, which are decided by the partitioning function. The

locations of the buffers on the local disk are sent to the master. Master knows location of all of the pairs and forwards these locations to the workers when assigning a reduce task.

 The master node tries to distribute the tasks uniformly over the cluster for load-balancing purposes. Otherwise, the whole operation might end up waiting for specific slow reducers. A typical and simplest partition function is to hash the key and use hash value modulo the number of reducers. The partition function might be more complex. The key responsibility of the partition function is uniform work distribution.

5. When master assigns a reduce task to a worker it passes location of data to be processed. The worker reads the data from the local disk of the map worker. When the whole data are read, the worker sorts it by the intermediate keys produced by the map worker. The sort operation is performed in order to group the entries with the same key together. This step is called shuffle. Many different keys may be mapped to the same reduce task, that is why sort is needed. Sorting is done using an external sort if data does not fit in the memory.

6. Then, the reduce worker begins to iterate over the sorted intermediate data. For each unique key, the key and the corresponding set of values is passed to the reduce function defined by the user. After each function call, output of the reduce function is appended to an output file for each reduce partition.

7. When the entire map and reduce tasks have been completed, the master nodes sends signal to the user program that executed the mapreduce call. Then, the user program wakes up and continues to execution.

Graphical representation of MapReduce processing steps is given in Figure 2.2.MapReduce frameworks may have different implementations. The most common frameworks are Apache Hadoop and Apache Spark.

Apache Hadoop is an open-source MapReduce framework written in Java for distributed storage and processing of very large data sets on computer clusters [4]. Hadoop assumes hardware failures are very common and focuses on automatic handling of such failures. Hadoop has two main components. The first one is storage

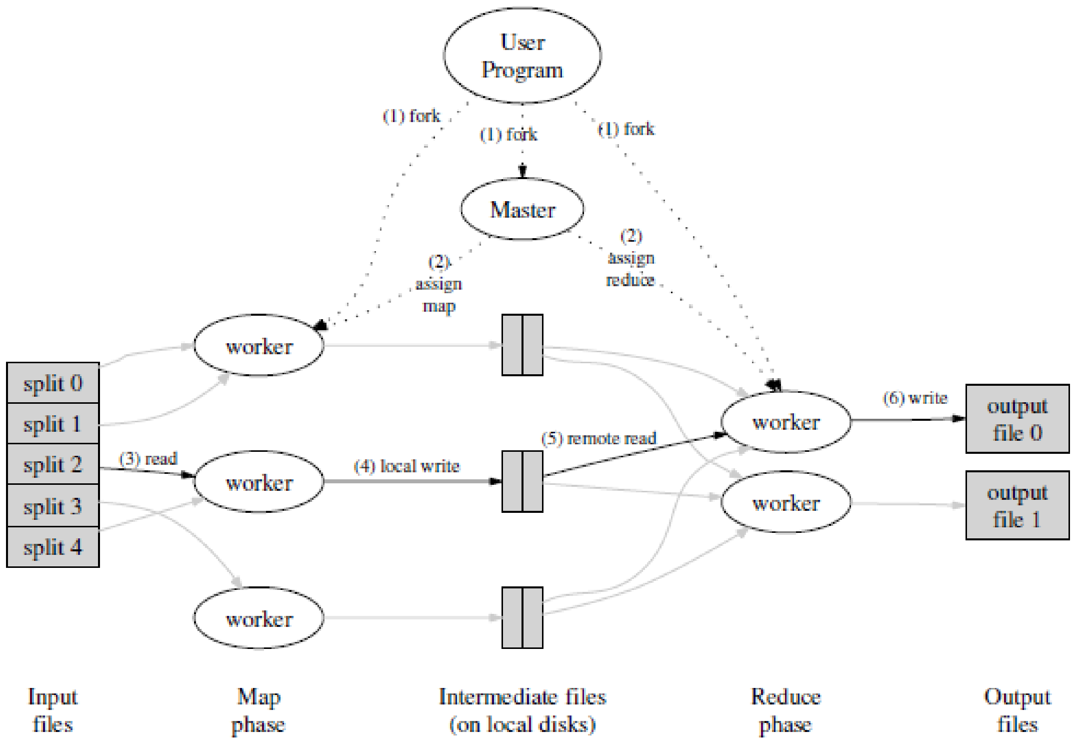


Figure 2.2: Domain Engineering and Application Engineering [22]

part (HDFS) and the second one is processing part (MapReduce). Processing steps of Hadoop are the same as the common MapReduce steps described above.

Apache Spark is another open-source framework for cluster computing [7]. While Hadoop performs two-stage MapReduce operations on disk, Spark processes data completely in main memory. Therefore, Spark provides a lot of performance gain for certain applications. In some situations where data does not fit in main memory, Spark can process some of the data on-disk. Unlike Hadoop, Spark does not write result of every map task to the disk. Processed data stays in the memory until commanded otherwise by the application. This difference is the main reason of speed difference between Hadoop and Spark. Spark uses a cluster manager and distributed storage system. Standalone mode (native Spark cluster), Hadoop YARN, and Apache Mesos are supported for cluster management. For distributed storage, Spark provides HDFS, Cassandra or Amazon S3 integration. Spark has SQL and Streaming components as well. Spark SQL provides support for structured data and enables to connect to an Apache Hive cluster and use it as data storage. Spark streaming module enables spark to process streaming data. It works by applying mini-batch operations over incoming data in very small intervals. Thanks to this design, the code written for batch operations can be used for stream analysis as well.

To summarize, the most important difference between Hadoop and Spark is Apache Hadoop processes the data on a distributed HDFS, while Apache Spark imports data into memory and performs processing completely in memory.

2.4.4.2 Stream Processing

Stream processing is a paradigm designed to automate the task of analyzing and acting on real-time streaming data. Stream processing paradigm deals with data while it is moving, before it has been persisted on a storage unit.

As the number of intelligent and data generating devices increase rapidly, data amount sent to the servers and data flow speed increase dramatically. In the meantime, technology requires incoming data to be analyzed as it arrives, almost in real-time. Stream processing frameworks enables us to do this kind of analysis.

In traditional computing, the data are first stored and indexed, then processed by queries. Stream processing takes flowing data while it is on its way to the server and applies analysis and transformations before it is written to the disk. Depending on the result of each analysis, system might take an action almost in real time against pre-defined situations.

Stream processing nodes can be configured to work like a pipeline or a single node. Some nodes may take action depending on the characteristics of incoming data, while some others transform data from one form to another. Output of a node may be input to another node or it can be directly written to the disk. At the end of the pipeline, data analysis is completed and data can be persisted on the disk. Here is a visualization of data streaming pipeline:

Stream processing is suitable for applications with following characteristics:

· Computation intensity (high ratio of I/O operations)

- Allowing parallel processing of data.
- Data are continuously fed from producers to consumers and data pipelining can be applied to flowing data.

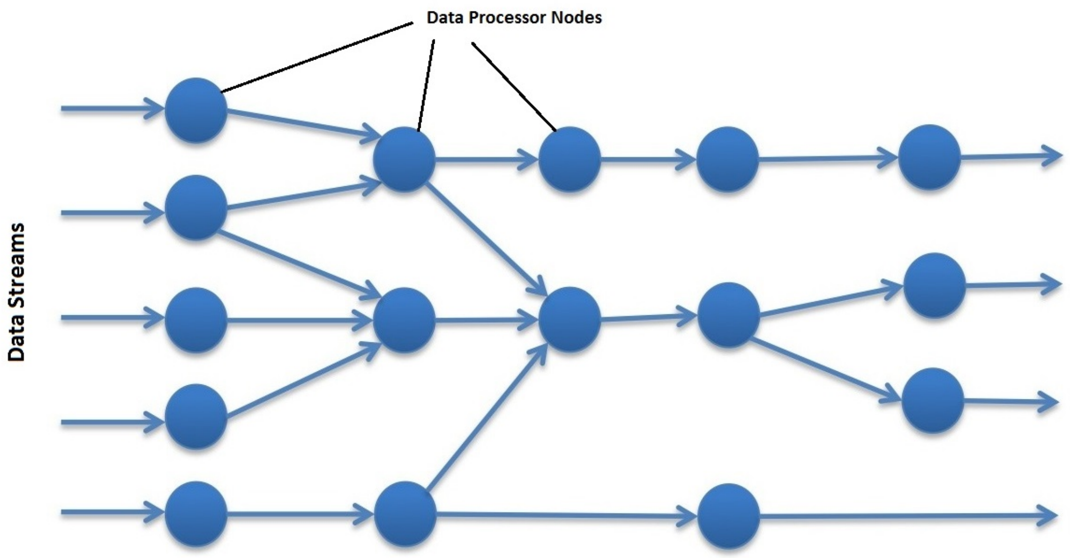


Figure 2.3: Domain Engineering and Application Engineering (adopted from [11])

Stream processing techniques are often applied to applications that process continuously flowing time-series data. Today, this technique is applicable to almost every industry, assuming as IoT technologies are going to develop rapidly and volume, variety and velocity of the data will increase dramatically. Stream processing methods are used widely in following industries where data flow is continuous: network monitoring, intelligence and surveillance, risk management, e-commerce, fraud detection, pricing and analytics, market data management and many more.

The most famous and common stream processing tools are Apache Storm, Amazon Kinesis and Apache Spark. Apache Spark is explained briefly in Section.2.4.4.1 as it has both MapReduce and stream processing modules.

Apache Storm is a computation framework designed to work on a distributed cluster [8]. Custom created ‘spouts’ and ‘bolts’ are used to define data sources and data processors, respectively, in order to allow distributed processing of streaming data. Topology of spouts and bolts is in the shape of a directed acyclic graph. Spouts and bolts are the vertices of the graph. Edges of the graph are named streams and they carry data from one node to another. Spouts and bolts work together to create a data transformation pipeline. A spout is source of streams. It reads data or generates somehow and emits the data as a stream. A bolt can consume any number of streams. It does some processing and then probably emits new streams. If a complex transformation needs multiple steps, each transformation is defined as a separate bolt and they are pipelined. As data model, storm uses tuples. A tuple can be defined as a named list of values. A field of a tuple can be any object. At higher level, storm can scale to massive number of emitted messages per second. In order to scale up, only new machines needs to be added to the cluster. Moreover, storm guarantees no data will be lost. Anything enters the cluster gets processed. When a bolt processes a tuple, it sends acknowledgement to its emitter.

Amazon Kinesis is a fully managed service for real-time processing of streaming data at massive scale [1]. Interface of Kinesis is relatively simple. User purchases throughput from Amazon and as long as the customer pays, requested performance is provided. Data producers puts data into Kinesis. The base stream unit of Kinesis is shards. Each shard provides fixed read/write throughput. Data are put into Kinesis by partition keys. Partition keys determine which data will go into which shard. If there is no partition key provided by producer, data goes into any shard. As data enters the shard, a sequence number is assigned by Kinesis. This may be used by consumer to determine the order of the data entering Kinesis. When reading data, readers read data by partition numbers. So each consumers consume data of a single shard. Maximum size of a single data unit can be 1 MB. There is no storage limit in Kinesis, just time limit. Data does not stay forever, a consumer needs to process it before it is removed from the stream. Data records are accessible for only 24 hours from the time they are added to a stream.

CHAPTER 3

LITERATURE REVIEW

A SmartGrid is an electrical grid that consists of various smart meters, renewable and efficient energy resources. SmartGrid is operated and watched remotely by computer-based systems. In domain of IoT and SmartGrid, a lot of researches have been done and are being done. They are mostly focused on Cloud Computing frameworks and their applications for Smart Grids. In this research, we will be studying HVAC EOS. These systems are similar to Smart Grid applications at higher levels and they are a subset of a Smart Grid. In both of them, there are large number of sensors deployed in the field, large amount of data flows from these sensors and these data creates the BigData. These data are analyzed for different purposes and using these data, forecasting is performed about future. However, the difference is places of the sensors and details of the analysis and forecasting algorithms. In Smart Grids, sensors are deployed over the electric lines while in HVAC EOS they are installed on devices running in a building. Smart Grid applications detect anomalies on the electric flow while HVAC EOS applications detect over-consumption of devices of the building and optimizes them. Therefore, a software architecture defined for a Smart Grid can be applied to HVAC EOS with small modifications. Thus, Smart Grid studies in the literature will be a reference for us along with HVAC and BigData related studies.

As a low level research, Sapna Tyagi et al. [40] studied a computing infrastructure for IoT data processing. Her target area was architectural and major challenges of massive data generated by IoT. She first examines the characteristics of IoT data. Then, present challenges of handling such large data sets. After analyzing the data, she describes some of the big data related technologies such as Apache Hadoop & NoSQL, Apache Hive, Apache Pig, and Apache Mahout. As the output of the study, she first demonstrates the proposed data transformation model for incoming data. Then, she proposes the main framework for managing the IoT data after transformations are applied.

Zhiming Ding et al. [23], worked on a topic similar to Tyagi's topic. He proposed a Sea-Cloud-based Data Management mechanism. First, he analyzes the key challenges of managing the IoT data. Then, presents an overview of the SeaCloudDM architecture and describe the database model. Different from the most of the IoT based researches; he uses a relational database along with a key-value store. After that, he describes data storage mechanism. Lastly, he discusses implementation issues and performance evaluation of the system. He calls the data receiving and storing layer as "sea-computing" since it works as counterpart of the "cloud computing".

While previous researchers worked on low level characteristics of BigData, Li et al. [30], proposed a higher level IoT data management solution called IOTMDB based on NoSQL. She analyzed the IoT data characteristics and proposed the solution that supports storing massive and diverse data. She discussed storage strategies. Moreover, she also analyzed and proposed data sharing mechanisms rather than just representing data. Finally, she studied index for stored data, query needs of an IoT system and proposed query syntaxes accordingly.

Besides these researches, there are a lot of researchers focusing on higher level cloud architecture design for Smart Grid applications. For example, Melike Yigit et al. [42] worked on cloud computing systems and their application to the Smart Grids. First, she focused on Smart Grid architecture and its applications. Then, she moved on to the Cloud Computing architecture while discussing opportunities and challenges of using Cloud Computing for Smart Grids. After the architecture, Cloud Computing is introduced in terms of efficiency, security and usability for Smart Grid applications. Technical and security analysis of Cloud Platforms' are presented. Lastly, current Smart Grid projects based on cloud services and open research issues are described. According to her, some of the issues are: Security framework for SG applications, Efficient Streaming with Clouds, Defining Communication Protocols and a Model for Network Utilization and Timely Demand Response.

As another research topic on Cloud Computing and Smart Grids, Sebnem Rusitschka et al. [33], presented a data management model for Smart Grids. She first analyzed popular Smart Grid use-cases. She stated that since most of the use-cases require collaboration of organizations, network operators, energy service providers and end users confidentiality and privacy in the design is at top priority. Then, based on the use-cases, she proposes a data management model. She designs the model based on the core characteristics of Cloud Computing such as distributed data management for real-time data collection, parallel processing for real-time information access, and omnipresent access.

Authors of [24] presented a Smart Grid information management paradigm using Cloud Computing. They have analyzed gains of using Cloud Computing for information management in Smart Grids. Then, they proposed a model to connect Smart Grid and Cloud Computing domains. Finally, four application scenarios were presented.

Some researchers focused on capabilities of current Cloud Computing platforms and their use-cases in Smart Grid domain. For example, Kenneth P. Birman et al. [20], analyzed needs of the Smart Grid technology and capabilities of current Cloud Computing platforms. First, he presented needs of the Smart Grids with examples that show large-scale computing has a key role in the smart power grid. Next, he described the current state of the Cloud Computing and examined its abilities if it can fulfill Smart Grids' needs. Then, he discussed issues about using the cloud for building the smart grid. He pointed advantages and disadvantages based on the current state of the industry. He grouped smart grid computing into three categories and discussed their needs. Those categories are: Applications with weak requirements, Real-time applications, Applications with strong requirements. He pointed that today's cloud lacks the technology for the last category. Finally, he discussed a few specific and important issues of cloud computing and smart grids and analyzed them deeper. As a result, he introduced a problem: the gap between today's cloud technology and real needs of a Smart Grid. He leaves the solution to future work and presents a research agenda.

These researches mostly focus on Smart Grid and Cloud integrations and capabilities of cloud platforms for Smart Grid applications. About software perspective of Smart

Grids, Yogesh Simmhan et al. [35] built a cloud-based software platform for data-driven analytics and he shared his experiences. In the research, he described methods he used for data ingestion, analyzing and forecasting. In the paper, he mostly focused on detailed algorithms and implementation methods rather than used tools and how the algorithms were applied to these tools.

In another research of Yogesh Simmhan et al. [36], they approached Smart Grid and Cloud Computing from another view. While other researches focus on technical details, he analyzed security and privacy issues of Smart Grid architectures on different cloud environments. He analyzed user, data, application and platform characteristics that have effect on security and privacy of Smart Grid applications on Cloud Computing platforms. He classified the platforms according to their risk factors and presented to the audience to reveal the risks.

In another research, researchers focused on sensor networks and energy analysis tools on cloud. Authors of [32] studied the integration of an in-building sensor network with a distributed Cloud environment. First, they described the development and use of sensor clouds. Then, they examined EnergyPlus [14], which is a tool for running energy simulations. They expressed how to use Cloud Computing in order to efficiently run and deploy EnergyPlus simulations and store its output. Finally, they described how to build such a sensor cloud application using a CometCloud [10] implementation that collects data from a real pilot building. While focusing on a single building application, they generalized their application for similar scenarios.

Demand-Response systems are an important subsystem of Smart Grid systems. On this topic, Authors of [29], proposed a cloud-based demand response (CDR) architecture for fast response times in large scale systems. CDR supports data-centric communication, publisher/subscriber and topic-based group communication rather than master/slave communication in order to make the CDR secure, scalable and reliable. They also proposed two distributed messaging algorithms. In the study, they first described how does CDR satisfies the requirements of security, scalability and reliability. Then, they discussed optimization algorithms for scalability, efficiency and speed. Then, they proposed the two algorithms that improve the convergence speed while keeping the messaging overhead the same.

Aside from these high level architectural studies, there are several studies that work on performances of NoSQL databases for BigData storage. For example, Rohan Narde [31] studied and tested performances of several NoSQL databases. He set up different levels workloads for NoSQL databases and tested some of the popular databases under these workloads. In the end he presented the results.

In a similar study, Bogdan George Tudorica et al. [39] tested and compared NoSQL databases from both qualitative and quantitative points of view. He presented features of NoSQL databases, installation size measurements and performance measurements. In the end, he presented test results.

All of the researches above seek solution to BigData storage and processing problems in a high architectural perspective or they describe or compare solutions for one subsystem only. They define what kind of cloud can be used under what conditions, analyze use-cases of cloud configurations, what type of components will be used at what point and measure performance of databases only. However, they do not describe or define complete software solutions that provide functionality for all needs of an HVAC EOS. None of them introduce a BigData management solution as a software architecture. For example, some of the researches describe a cloud architecture but do not mention which tools will be used and how they will be configured, or how the database will be designed and optimized for different types of data sets. This research aims to introduce an insight and propose a relatively low-level design and an optimal software architecture for BigData management of HVAC energy optimization systems.

CHAPTER 4

A REFERENCE HVAC ENERGY OPTIMIZATION SYSTEM

This chapter will describe our reference HVAC energy optimization system in detail. HVAC stands for heating, ventilation, and air conditioning. An HVAC energy optimization system's purpose is to determine the working parameters and scenarios of the HVAC equipment in order to minimize the cost and maximize the efficiency. The workflow of an HVAC energy optimization system is shown in Figure 4.1

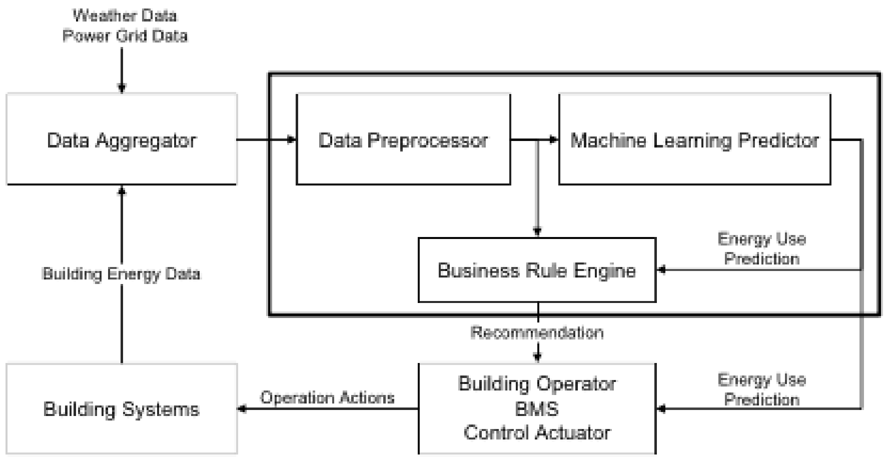


Figure 4.1: Workflow of an HVAC EOS

An HVAC energy optimization system is a good example for a BigData system. It collects data from lots of sensors, some of which collect data about the environment and some others collect data about the devices running in a building. These sensors generate the BigData of our system. This BigData has to be stored efficiently, so this is the first challenge that the system encounters, and also a typical one for Big-Data systems. After data is stored, it has to be queried and analyzed by concurrent processes. This processing is not performed easily and efficiently with traditional systems. Therefore, this is the second challenge that the system has to cope with. Again this is also another typical challenge that BigData systems encounter. Consequently, an HVAC energy optimization system is an appropriate reference for us to analyze BigData.

In the following sections, high-level operational and performance requirements of the system, details of data flow and detailed steps of data processing will be given. In addition, data types used in an HVAC energy optimization system are defined.

4.1 Design Requirements

The architecture proposals of the system are going to be described in Chapter 5. However, interaction of the subcomponents with each other is already implemented and depends on the tool, not the system design. However, the system has two points that interacts with the outer world. The first one is the data visualization component, from which user can interact with the system and display the data submit jobs. Communication of the client (user) and the system can be implemented in anyway. It may be an XML or REST web service or even a socket. However, the other interaction point of the system is not that flexible. The other point is the interface between the sensors on the field and the HVAC EOS. Most of the sensors in the market communicates through either MODBUS over TCP/IP or HTTP protocols. Therefore, the HVAC EOS has to be able to communicate with both protocols in order to collect data from the field.

4.2 System Attributes

We are going to discuss qualitative attributes of the system in three main topics:

Reliability of the System: The system has to display right data to the user 99% of the time and do consumption and cost estimations correct up to 95% of the time. The details of estimation operation is out of the scope of this thesis. It is the job of the machine learning algorithms that run on the system.

Availability of the System: The system has to respond to the user 100% of the time. The result may have a few missing data from last few minutes since it will not affect overall consumption and cost analysis too much. Moreover, the system has to be able to collect data from sensors fast enough to not let any data to be lost or overwritten by the sensor.

Security of the System: The system must not allow unauthorized access. In a cloud environment, the cloud service provider has to guarantee this and in a local cluster, system admin has to ensure that only authorized people can access the data. In either case, details of the requirement is out of the scope of this thesis.

4.3 System Operation

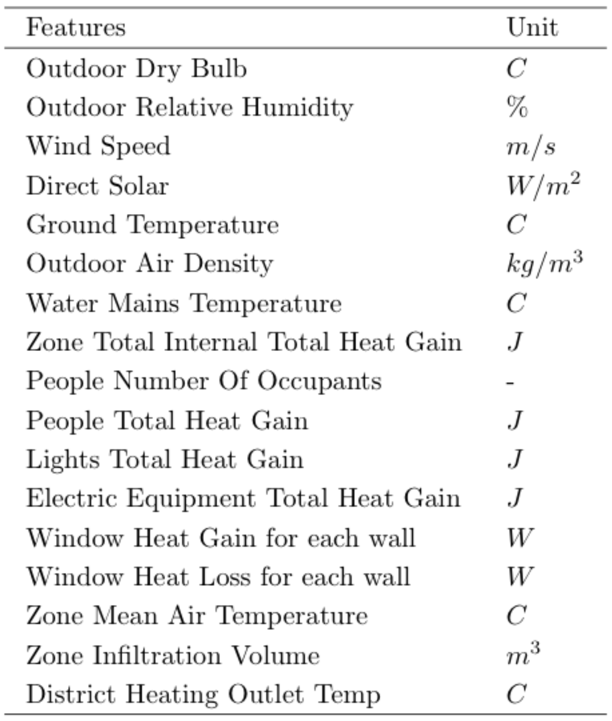
This section will explain operations performed on the system and performance requirements of them. In the further chapters, different system architectures will be suggested for this system.

4.3.1 Operation 1: Static Building Energy Performance Simulation

This simulation is performed with EnergyPlus. EnergyPlus is an energy simulation tool that enables building professionals to optimize the building design to use less energy and water [14]. EnergyPlus calculates heating and cooling loads necessary to maintain thermal control set-points and the energy consumption of HVAC equipment. Program gives energy consumption and performance estimations as output.

As first step of the operation, the building is modeled with EnergyPlus simulation tool. Inputs of the program are as follows:

- Weather conditions, including dry bulb temperature, relative humidity, global horizontal radiation, wind speed, etc.
- Building structure characteristics, including shape, capacity, orientation, window/wall radio, fenestration/shading, building materials, thermal zones, etc.



Features	Unit
Outdoor Dry Bulb	C
Outdoor Relative Humidity	%
Wind Speed	m/s
Direct Solar	W/m^2
Ground Temperature	C
Outdoor Air Density	kg/m^3
Water Mains Temperature	C
Zone Total Internal Total Heat Gain	J
People Number Of Occupants	-
People Total Heat Gain	J
Lights Total Heat Gain	J
Electric Equipment Total Heat Gain	J
Window Heat Gain for each wall	W
Window Heat Loss for each wall	W
Zone Mean Air Temperature	C
Zone Infiltration Volume	m^3
District Heating Outlet Temp	C

Figure 4.2: Inputs of EnergyPlus

- Occupants' behaviors, such as number of occupants, their leaving and entering time, thermal comfortable set points, ventilation, etc.
- Inner facilities and their schedules, like lighting and HVAC system, plug load.

Detailed inputs of EnergyPlus are shown in figure.4.2. After the simulation is completed, the program outputs energy consumption and performance predictions for the building for different periods. These data will be stored in the database and will be used in decision algorithms. Here is a simple pseudo-code of the operation performed for static simulation:

```
function staticSimulation {
  buildingData = readBuildingCharacteristics()
  weatherData = readWeatherForecast()
  occupantData = readOccupantData()
  deviceData = readBuildingDeviceInformation()
  startStaticSimulation(buildingData, weatherData,
              occupantData, deviceData)
}
```

Use-case diagram for the operation is given in Figure 4.3 and sequence diagram is given in Figure 4.4.

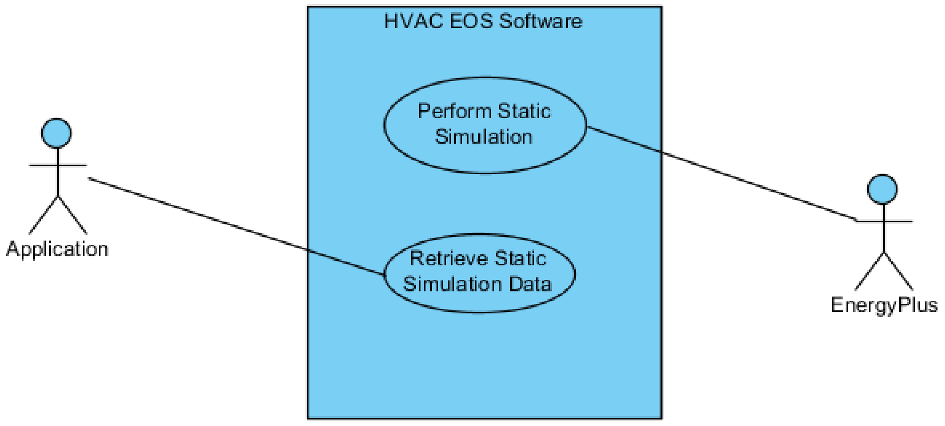


Figure 4.3: Use-Case Diagram for Static Simulation

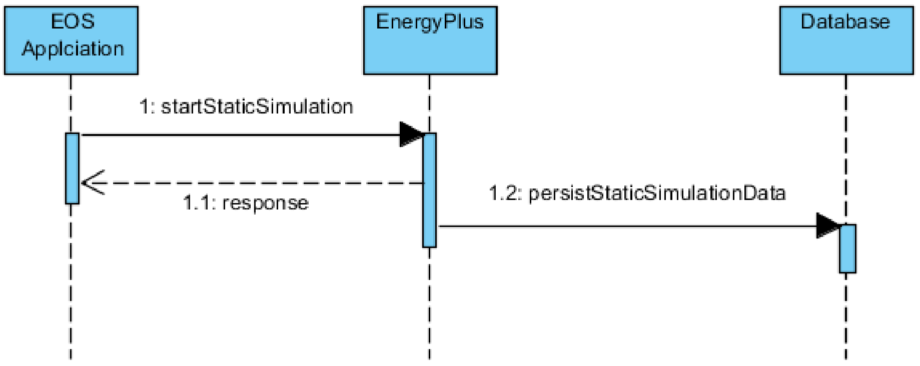


Figure 4.4: Sequence Diagram for Static Simulation

Performance Needs For these operations there are no strict performance needs since they are performed only once when building is first introduced to the system. A typical simulation lasts for about 3 to 4 hours.

Performance Requirements of Queries Since this operation is performed only once for each building, there are no strict performance requirements for queries. As all of the required data is static information, there will be only read operations from the database tables or data storage medium that store these data and queries will be performed fast enough. All of the four queries used in this operation shouldn't last more than 4 or 5 seconds.

4.3.2 Operation 2: Sensor Data Processing

After the simulation is performed, rest of the operations is based on the incoming sensor data. These operations are performed continuously on predefined periods or on user requests, different from the simulation.

4.3.2.1 Step 2-a: Data Preprocessing

The raw data are not ready for constructing forecasting model because some values may be in wrong format. The data preprocessor receives various data streams from the data aggregator and restructures them so that all the data fit the format required by the Machine Learning prediction and business rule engine. The following preprocessing tasks have to be performed on incoming data:

Fill missing data: The lack of some information may decrease the predictive efficiency of the forecasting model. Recent data are scanned and the missing data are filled by prediction.

Change data format: Data are collected every 5 minutes. So, an appropriate aggregation function should be used to store aggregated data. For example, aggregate data need to be stored aggregated hourly, daily and weekly.

Check data and system quality: In order to ensure that the machine learning system is reliable all the time, the internal and external conditions, control actions, and the results of those actions should be evaluated using an automated evaluator. The automated evaluation system receives data at multiple stages in the system workflow. The evaluator employs intelligent real-time data quality analysis components to quickly detect data anomalies (malfunctions of digital thermostats that interfere with temperature reading or introduce variances from normal expected HVAC set-points) and gives feedback to building management, who can then respond appropriately.

A visual representation of data flow is shown in Figure 4.5

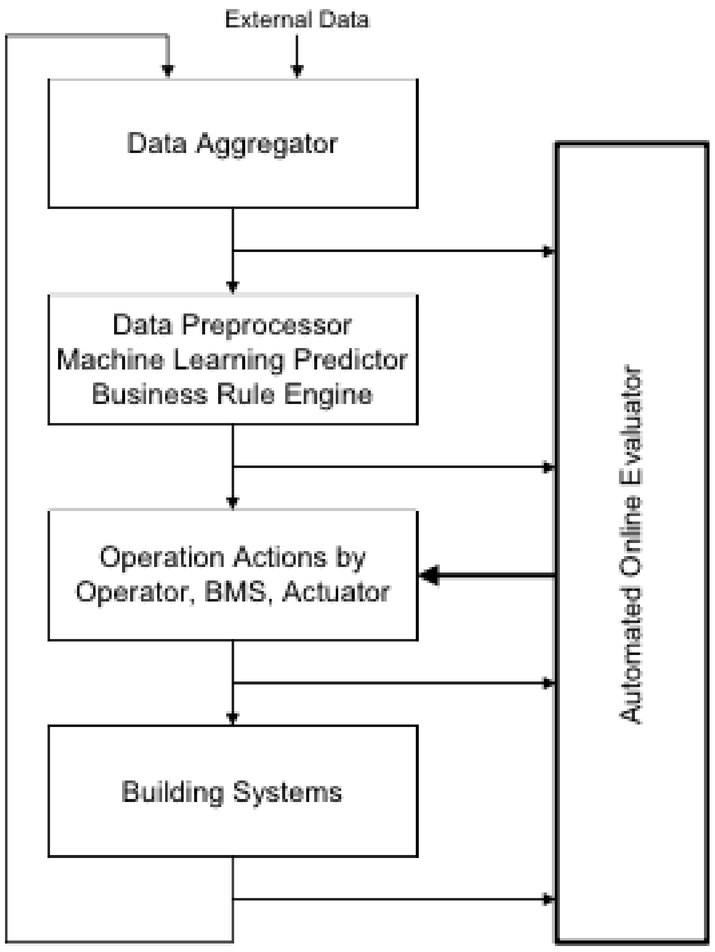


Figure 4.5: Data Flow of a HVAC EOS

Here are the parameters and techniques of real-time analysis of data and system quality:

Thresholds: The thresholds define the normal working range of the specific data points. If the data reading exceeds the thresholds at either the lower or upper bound, the data record will be flagged as anomalous and a corresponding warning will be communicated back to the building operator.

Online Anomaly Detection Algorithm : Anomaly detection finds data instances that are unusual and do not fit any established pattern. It concentrates on modeling normal behavior in order to identify unusual data points. This system processes the continuously updated data-streams to detect anomalies.

Visualization: Visualization provides an easy way to obtain additional verification of a data anomaly. This component is also a useful communication channel to help building manager to understand where issues are arising.

Use-case diagram for the preprocessing operation is given in Figure 4.6 and sequence diagram is given in Figure 4.7. In addition, here is the pseudo-code of the operations for data preprocessing performed for each data unit.

```
function performDataPreprocess(dataUnit) {
  isDataValid = checkDataValidity()
  if (isDataValid == false) {
    return
  }

  isDataMissing = checkIfDataMissingSinceLastDataReceived()
  if (isDataMissing) {
    fillMissingDataWithPredictions()
  }

  reformatData()
  saveData()
}
```

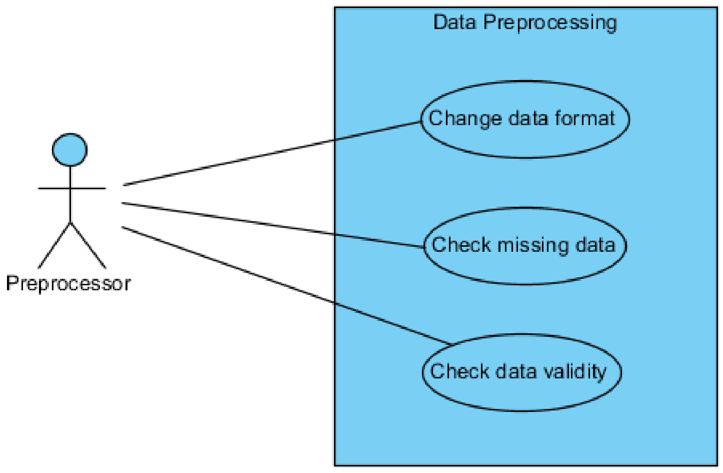


Figure 4.6: Use-Case Diagram for Data Preprocessing

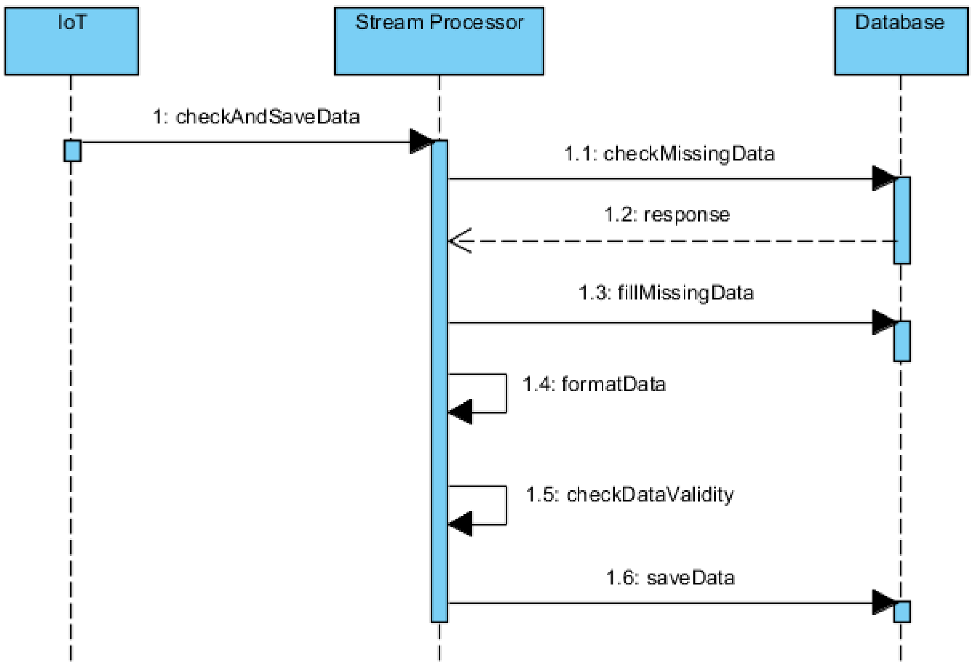


Figure 4.7: Sequence Diagram for Data Preprocessing

Performance Needs Any delay in this operation may lead to wrong or deficient calculations in the system since the main data of the calculations pass from this operation. Therefore, each data packet has to be preprocessed in less than 100 milliseconds after it enters the system.

Performance Requirements of Queries There is only one possible query in this operation and even it might not exist depending on the processing methodology. According to the performance needs of this operation, if database is queried, it has to return in less than 100 milliseconds.

4.3.2.2 Step 2-b: Watch Building Demand

Demand data of the building should be watched continuously in order to respond rapid changes in energy demand of the building. This operation is performed by taking some recent data into consideration, not only incoming data. First, the trend of the consumption is calculated using the incoming data and recent data. Then, decisions are made accordingly. Steps for this operation:

- Retrieve last day's demand data starting from work start time. Using this data, estimate demand of the following 15 minutes.
- Retrieve demand estimation for next 15 minutes from energy plus output.
- Decide one of the two estimations and send appropriate commands to BMS.

These data retrieval operations do not have to be database queries. If data are processed in the data stream, the summarized information about recent demand data can be kept on processing nodes. Use-case diagram for watching building demand operation is given in Figure 4.8 and sequence diagram is given in Figure 4.9. Here is the pseudo-code of operations for watching building demand:

```
function watchBuildingDemand() {
  lastDaysDemands = retrieveDemandDataOfLastDayFromRealData()
  realEstimations = doEstimationsForNext15Mins(lastDaysDemands)
  simEstimations = retrieveEstimationsFromStaticSimulation()

  decideNextEstimationAndTakeAction(realEstimations,
                                    simEstimations)
}
```

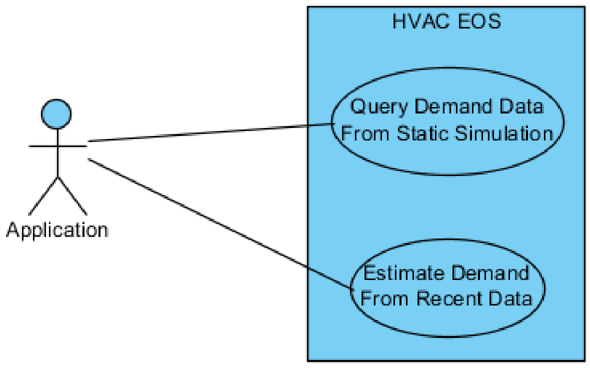

HVAC EOS
Query Demand Data From Static Simulation
Estimate Demand From Recent Data
Application

Figure 4.8: Use-Case Diagram for Watching Building Demand

Performance Needs Data for this operation are received every 5 minutes. Therefore, system needs to process and analyze demand data once every 5 minutes for each building.

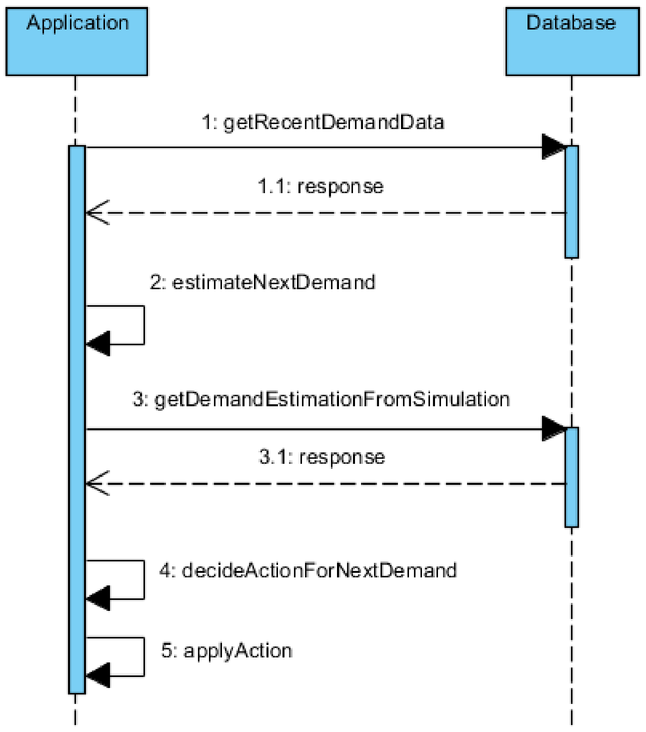


Figure 4.9: Sequence Diagram for Watching Building Demand

Performance Requirements of Queries In this operation, only demand data of last day are queried. Depending on the calculation methodology, data might be kept in application with no need of query. Data size of one day is about 288 data records for each sensor, which is quite low. Therefore, query needs to return in less than a second. If it takes longer, it may point to a problem in the system.

4.3.2.3 Step 2-c: Alarms and Notifications

All other analyzes are performed on the incoming data. Besides these operations, user of the system can define alarms and notifications to be triggered when a certain event occurs. An alarm or a notification occurs mostly when a specific value is above or under a defined threshold. Fast analysis and rapid notifications allow user to perform some set of energy-efficient actions and increase overall awareness of the user. Here is the pseudo-code of operations for Alarms and Notifications module:

```
function checkAlarms(dataUnit) {
  alarmsDefs = retrieveDeviceAlarmDefinitions(dataUnit.getDevice())
  for(def in alarmDefs) {
    if (checkIfAlarmTriggered(dataUnit, def) == true) {
      triggerAlarmAndSendNotification(def)
    }
  }
}
```

Use-case diagram for alarms and notifications is given in Figure 4.10 and sequence diagram is given in Figure 4.11.

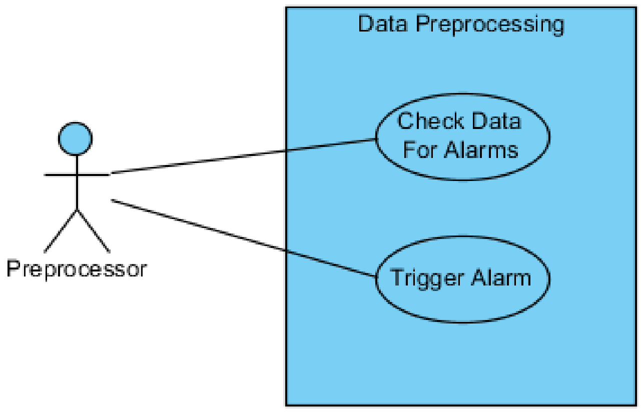


Figure 4.10: Use-Case Diagram for Alarms & Notifications

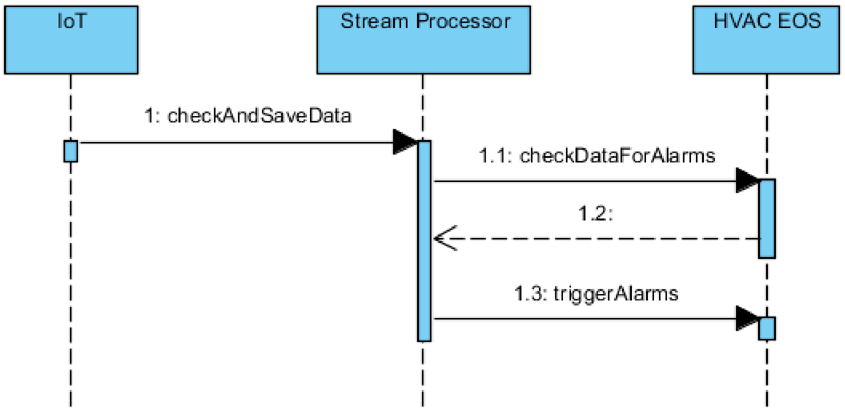


Figure 4.11: Sequence Diagram for Alarms & Notifications

Performance Needs Like data preprocessing operation, this one is also performed for each data record that enters the system. For each incoming data record, alarm check needs to be completed in less than 100 milliseconds in order to inform the user and take appropriate actions in the system since some alarms may be critical and require immediate action.

Performance Requirements of Queries For a sensor, there can be at most four to five alarm definitions typically. These alarms are either kept in the memory of the application or queried from the database depending on the processing methodology. If query is used, query has to return in less than 100 milliseconds.

4.3.2.4 Step 2-d: Forecasting

This operation's purpose is to make predictions about building's energy consumption in near future by using the past data to learn behavior of the building.

In order to learn energy behavior of the building, the forecasting module needs to learn three characteristics of the building. These are occupancy behavior, energy consumption of the building in the recent past and in a longer past.

First, occupancy behavior of the building has to be learned. Inputs of this operation:

- A static schedule data of the building (working hours and days of the building).
- Real-time occupancy data from sensors to detect how many people are in the building.
- Occupancy inputs from the BMS if there is one available.

Then, the forecasting module analyzes these inputs and learns the occupancy behavior and trend of the building. Next step is learning the thermal behavior of the building in recent past. Data set is kept small in this operation since thermal data may vary greatly among seasons. Steps for this operation:

- Last three months' air quality and comfort parameters are retrieved.
- HVAC equipment's' running schedules are retrieved for last three months.

- HVAC equipment's' energy consumption details are retrieved for last three months.
- These parameters are analyzed together to learn how did the building perform recently.

Next operation is to learn how the building performed in a wider history. Data set is larger in this operation in order to learn how the building responds to the environmental and seasonal changes. Steps are:

- Retrieve last one year's energy consumption for the building or device. Then, process it and do estimation for oncoming consumption of the building or device using Machine Learning techniques.
- Retrieve last one year's energy consumption data of the same day of the week as day that estimation will be performed. Then, process it and do estimation for oncoming consumption of the building or device using Machine Learning techniques.
- Lastly, retrieve outputs of energy plus simulation and its estimation.
- Compare all these three estimations and decide one or combination of them.

The details of these analysis and machine learning algorithms are out of the scope of this research.

Use-case diagram for consumption forecasting operation is given in Figure 4.12 and sequence diagram is given in Figure 4.13. In addition, here is the pseudo-code of operations for forecasting:

```
function forecast() {
  buildingSchedule = retrieveBuildingSchedule()
  occupancyData = retrieveOccupancyData()

  airParameters3m = retrieveAirParametersOfLast3Months()
  hvacRuntimeData3m = retrieveHVACRuntimeDataOfLast3Months()
  hvacConsumption3m = retrieveHVACConsumptionOfLast3Months()

  result3m = analyze3mData(buildingSchedule, occupancyData,
      airParameters3m, hvacRuntimeData3m, hvacConsumption3m)

  airParameters1y = retrieveAirParametersOfLast1Years()
  hvacRuntimeData1y = retrieveHVACRuntimeDataOfLast1Years()
  hvacConsumption1y = retrieveHVACConsumptionOfLast1Years()
  hvacConsumption1ysd =
      retrieveHVACConsumptionOfLast1YearsSameDayOfWeek()

  result1y = analyze1yData(buildingSchedule, occupancyData,
                    airParameters1y, hvacRuntimeData1y,
                    hvacConsumption1y, hvacConsumption1ysd)

  staticSimResults = retrieveStaticSimulationEstimationsNext1Year()

  analyzeAndDecideForecast(result3m, result1y, staticSimResults)
}
```

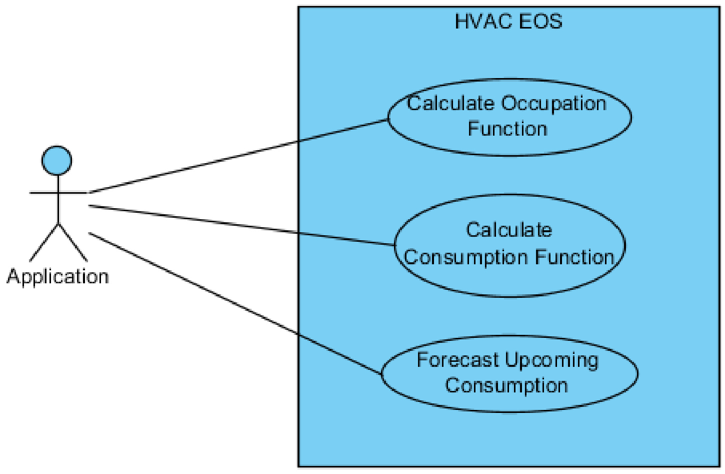

HVAC EOS
Calculate Occupation Function
Calculate Consumption Function
Forecast Upcoming Consumption
Application

Figure 4.12: Use-Case Diagram for Consumption Forecasting

In order to optimize the building control set-points, a mathematical model is developed. Then, historic building performance data of up to 3 months will be used in order to identify each variable in this model.

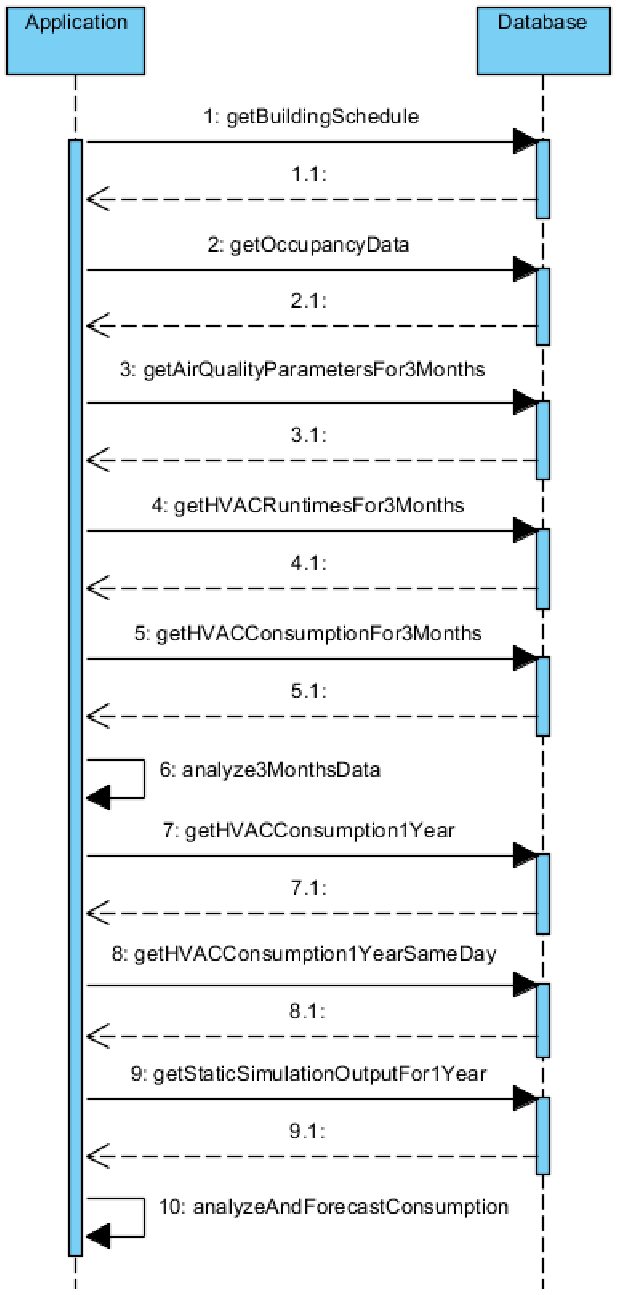

Application
Database
1: getBuildingSchedule
1.1:
2: getOccupancyData
2.1:
3: getAirQualityParametersFor3Months
3.1:
4: getHVACRuntimesFor3Months
4.1:
5: getHVACConsumptionFor3Months
5.1:
6: analyze3MonthsData
7: getHVACConsumption1Year
7.1:
8: getHVACConsumption1YearSameDay
8.1:
9: getStaticSimulationOutputFor1Year
9.1:
10: analyzeAndForecastConsumption

Figure 4.13: Sequence Diagram for Consumption Forecasting

Performance Needs This operation is performed 3 to 4 times a day for a single building. This operation has to be completed as soon as possible since all further calculations depend on the output of this operation. If this operation cannot complete quickly, calculations are performed with out of data.

Performance Requirements of Queries This operation requests at most data of last one year. This query has to return in less than 6 seconds. If it takes longer, real-time property of the system is hard to be maintained.

4.3.2.5 Step 2-e: BMS Optimization

For this operation, a supervisory control and optimization algorithm is used for HVAC systems. This algorithm aims minimizing energy consumption and occupant thermal discomfort. An accompanying mobile occupant comfort feedback tool is provided to building inhabitants. Feedbacks from this tool are used as input to the optimization algorithm and used in order to assess the impact of the optimized HVAC control strategy, to fine-tune the standard comfort model used in the optimization process and to inform users of the state of the HVAC system, such as when natural ventilation or economy modes are active. It allows occupants to submit feedback on their thermal comfort as well as satisfaction at any point.

The supervisory HVAC control and optimization system interfaces with the commercial building management and control system to read required data from HVAC zones and air handling units (AHUs) and to perform required control actions. It learns a model that can be used to predict zone conditions and comfort levels using information about HVAC power consumption and weather, which are used to discover the optimal power consumption schedule throughout the day in order to balance cost, energy and emissions while maintaining comfort. The control module then overrides zone or supply air set-points to balance the heating/cooling power supplied to each zone and to track the best power profile from the optimization. The system learns a model from historical data and uses weather forecasts and a given temperature set-point to profile and predict building zone conditions and thermal comfort.

In order to summarize the operation, the inputs are:

- AHU operational and energy consumption parameters for last twelve days.
- Occupancy forecast of the current day and occupancy values of last twelve days.

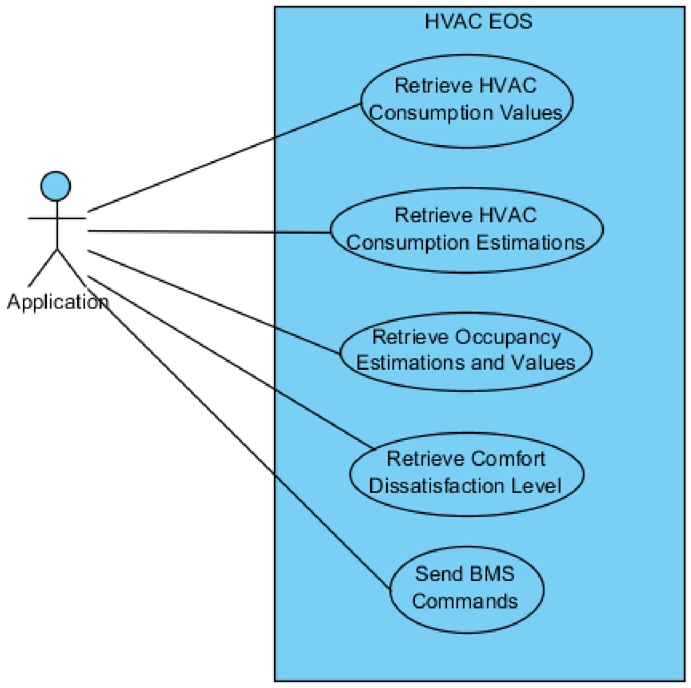

HVAC EOS
Application
Retrieve HVAC Consumption Values
Retrieve HVAC Consumption Estimations
Retrieve Occupancy Estimations and Values
Retrieve Comfort Dissatisfaction Level
Send BMS Commands

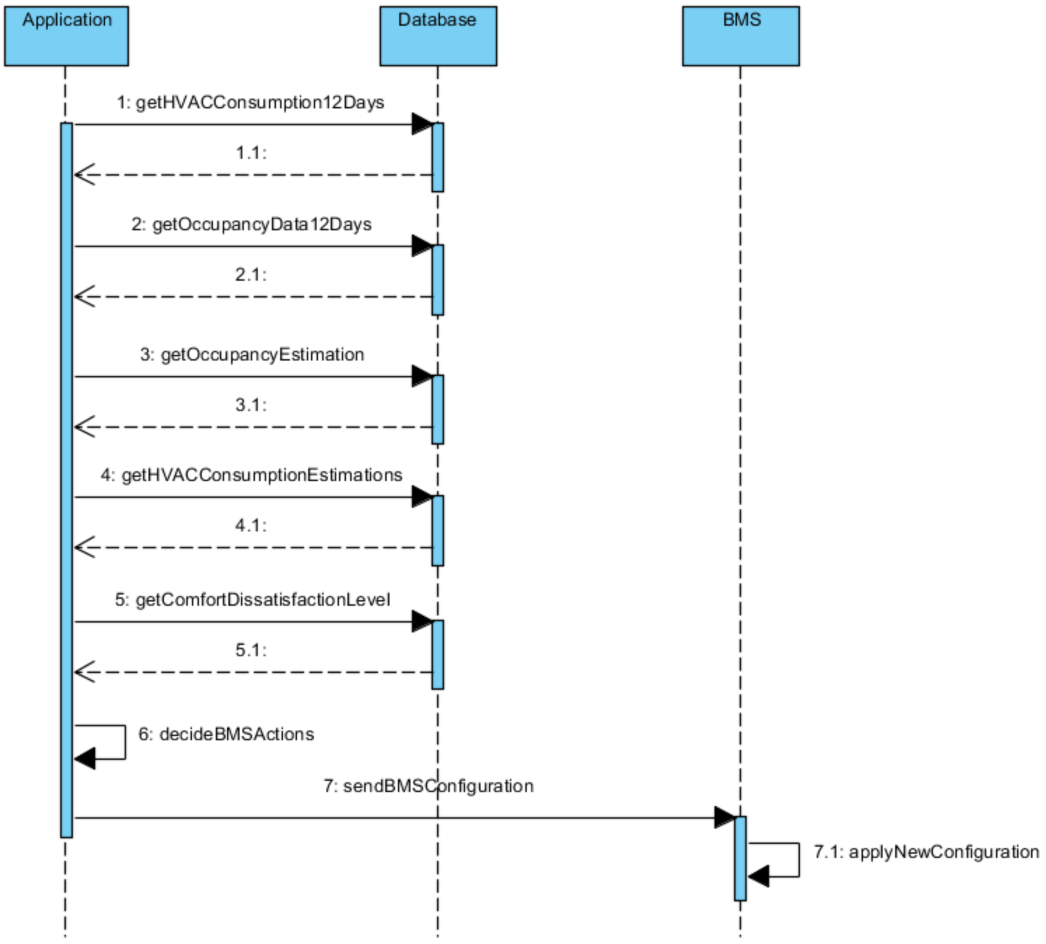
Figure 4.14: Use-Case Diagram for BMS Optimization

- Energy consumption forecast of the current day for upcoming hours.
- Comfort dissatisfaction level feedback from the occupants of the building.

Using these inputs, the BMS optimization algorithm decides on temperature and pressure set-points of AHU. It evaluates trade-offs between occupant satisfaction and energy costs and decide on values keeping a balance between these parameters. The detail of the algorithm is out of the scope of this research. Use-case diagram for BMS optimization is given in Figure 4.14 and sequence diagram is given in Figure 4.15. Here are the pseudo-code of the operations for optimizing the BMS:

```
function bmsOptimization() {
  hvacConsumption = retrieveHVACConsumptionOfLast12Days()
  occupancyData = retrieveOccupancyDataOfLast12Days()
  occupancyEstimation = retrieveTodaysOccupancyEstimation()
  hvacEstimations = retrieveHVACForecastForToday()
  comfortDissatisfaction =
      retrieveOccupantDissatisfactionDataForToday()

  bmsCommand = analyzeAndDecideBMSAction(hvacConsumption,
        occupancyData ,  occupancyEstimation ,  hvacEstimations ,
        comfortDissatisfaction )
    applyBMSCommand ( bmsCommand )
}
```



Application
Database
BMS
1: getHVACConsumption12Days
1.1:
2: getOccupancyData12Days
2.1:
3: getOccupancyEstimation
3.1:
4: getHVACConsumptionEstimations
4.1:
5: getComfortDissatisfactionLevel
5.1:
6: decideBMSActions
7: sendBMSConfiguration
7.1: applyNewConfiguration

Figure 4.15: Sequence Diagram for BMS Optimization

Performance Needs Similar to the previous operation, this one is performed 4 to 5 times a day for a single building. This operation has to be completed as soon as possible since all further calculations depend on the output of this operation. Until this operation is completed, calculations are performed with out of data.

Performance Requirements of Queries This query has to return in less than 6 seconds in order to maintain real-time feature of the system.

4.4 Data Used for Operations

There are several data types and sources that will be used for energy optimization. They will be described without deep details and specific structure. Each data unit holds either numerical data or time-stamps.

Sensor Data: This data type is the largest data set in the system. There may be various energy sensors deployed in a building. Even if the sensors sense the same data, they might be from different vendors and their data format may not be the same. Consequently, there will be data from lots of sensors and their column numbers will differ.

As a typical property of optimization systems, each sensor collects and sends data of last five minutes, every five minutes. Data may be cumulative or instantaneous depending on the type of the measured value.

Air Quality: This type of data are simpler than the sensor data table. It holds three properties of the air at a certain time. They are temperature, humidity and CO2 concentration. Besides these three data, there is also a timestamp accompanying these three numeric data.

Like the sensor data, this type of data are collected every five minutes. Data are instantaneous.

Working Schedule: This data set is a static one. Working schedule of the building is defined and updated manually in predefined periods. Working schedule means daily working hours and workdays in a week. Mostly there are data for seven days. If there is an exceptional day like national holidays, it is defined here as well.

This data set is managed by the building energy operator.

Occupancy: Structure of this data set is similar to the air quality data. Data are generated by occupancy data of the building at certain times. There are only two numeric values for each data unit, timestamp and occupant number.

Like other dynamic data sets, data is gathered every five minutes. Data are instantaneous.

Price Tariff: This data set is another static one. These data are used to gather energy prices. Prices may be different at various times of day. Price tariff data are used to minimize the total energy cost. This data set has three values for each unit, start time of a price tariff definition, end time of a price tariff definition and price of the energy in that interval.

This data set is managed by the building energy operator or an automated system that retrieves prices whenever they change. Data are instantaneous.

CHAPTER 5

PROPOSED ARCHITECTURES

For the described energy optimization system, software architecture alternatives will be proposed in this section. First few architectures can handle only small or moderate amount of flowing data and their processing. However, as the sensor nodes in the system increase, data amount and velocity may increase exponentially. In order to cope with this amount of data, at the end, a completely scalable system will be proposed. Each architecture-set introduces a missing feature or improves a feature of previous architectures.

5.1 Architecture with only NoSQL Database and Data Processor

This is the simplest and the most straightforward scenario. Incoming data are directly written into the database. Three alternative architectures will be proposed without major architectural changes. Only interchangeable parts are tools.

Solutions in this section can handle large amount of data sources only if incoming data are written directly to the database without any processing. The performance is limited by performance of the node that retrieves and writes data.

5.1.1 System with Key-Value Store and Manual Data Processing

In Figure 5.1, a high level design of this proposal is visualized. In this architecture, a key-value store is used as database. In particular, Amazon's DynamoDB. While creating tables to store sensor data, hash and range type primary key is used. A

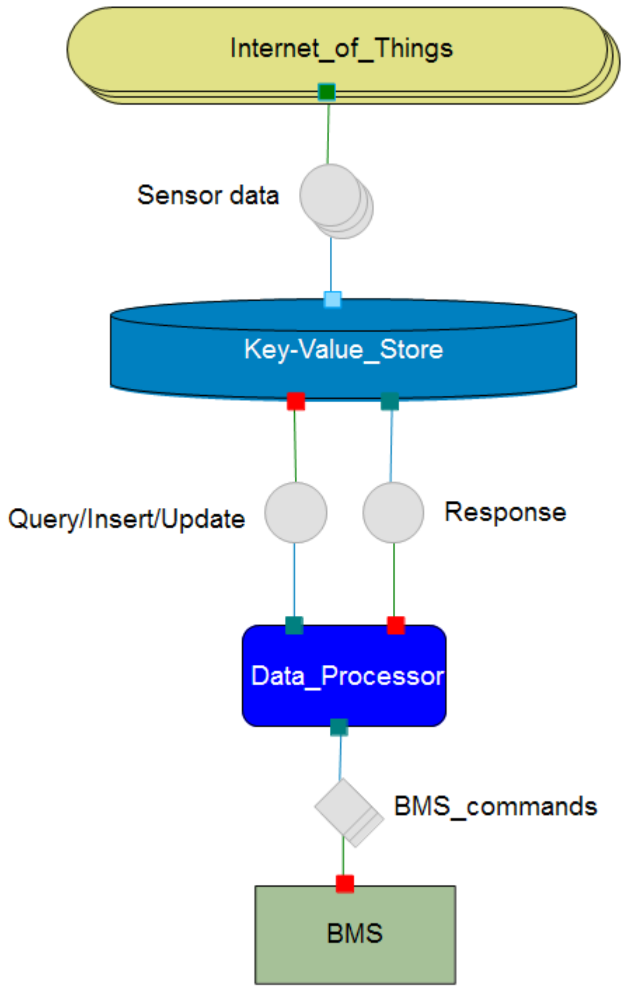


Figure 5.1: System Architecture with Key-Value Store and Manual Data Processing

unique sensor id is used as hash key in order to store data of same sensor together. As the range part of the key, timestamp is used. In order to keep timestamp in a single column, a string value created by concatenating of year, month, day, hour and minute values. In order to be able to sort data by date, concatenation is done in stated order.

A data processor application retrieves data from database, performs analyzes and processing. Afterwards it writes result of the processing back to the database and takes appropriate actions, if necessary, on automation system. This application is an implemented manually, it is not a commercial or open source application.

This architecture is suitable for a system with large number of data sources and small number of consumers or development purposes. As consumer number increases, this system will begin to suffer from performance problems. Moreover, this system cannot handle queries that run over very large data sets and return large amount of data. In addition, data processing performance of this system is not high since processing is performed on a single application. This system is not suitable when a database row contains too much data because DynamoDB has an upper limit on row size. On the other hand, this system has almost no maintenance cost since everything is managed by Amazon as long as cost is paid. This may be a big plus if user does not want to deal with database maintenance.

5.1.2 System with Key-Value Store and Distributed Data Processing

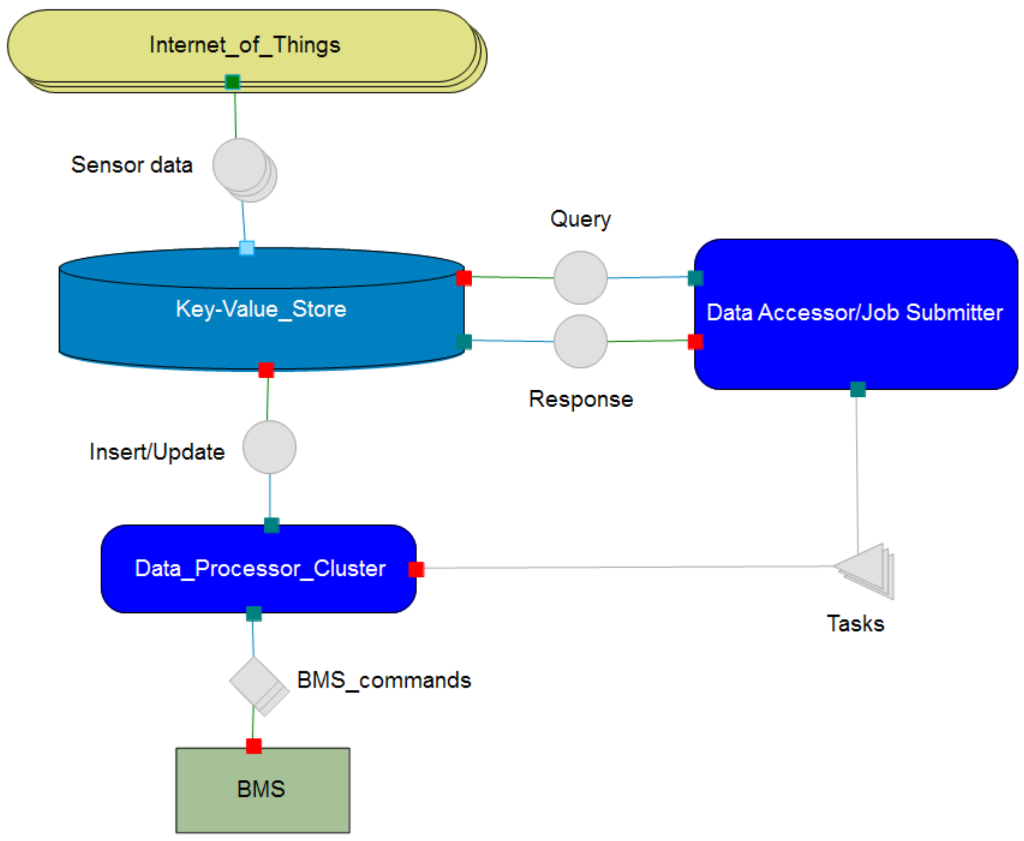


Figure 5.2: System Architecture with Key-Value Store and Distributed Data Processing

In Figure 5.2, a high level design of this proposal is visualized. The database part is the same as previous architecture. Amazon's DynamoDB is used as database. The primary key of the table is hash and range type. A unique sensor id is hash key and timestamp is the range part of the primary key. In order to keep timestamp, a string is built from concatenation of year, month, day, hour and minute values, so that data can be sorted by date.

In this architecture, data processor is not a user written application. A MapReduce framework will be used. In particular, Apache Spark will be used. Data processor is not a single node. It is a cluster of computers that run as Spark nodes. However, still there has to be a node that submits works to the Spark cluster. A drawback of this architecture is, Apache Spark does not have built-in integration or library for DynamoDB integration since DynamoDB is proprietary service of Amazon and Spark is open source. The work submitter node has to first get the information data from DynamoDB. Then, it has to distribute the data over the Spark cluster. This operation is done with 'parallelize' method in Spark. With this method call, any local data can be distributed over the cluster. However, if the required data amount is too large, this operation may be slow since there will be two time transmission of the whole data. First, the whole data are retrieved from DynamoDB, then the whole data are transmitted to the nodes.

This architecture is suitable for a system with large number of data sources and small number consumers or development purposes. As consumer number increases, this system will begin to suffer from performance problems. Moreover, this system cannot handle queries that run over very large data sets and return large amount of data. Data processing performance of this system is better than the previous solutions since there is a Spark cluster performing the processing, not a single application. This system is not suitable when a database row contains too much data because DynamoDB has an upper limit on row size. On the other hand, this system has small amount of maintenance cost since everything except Spark cluster is managed by Amazon as long as cost is paid.

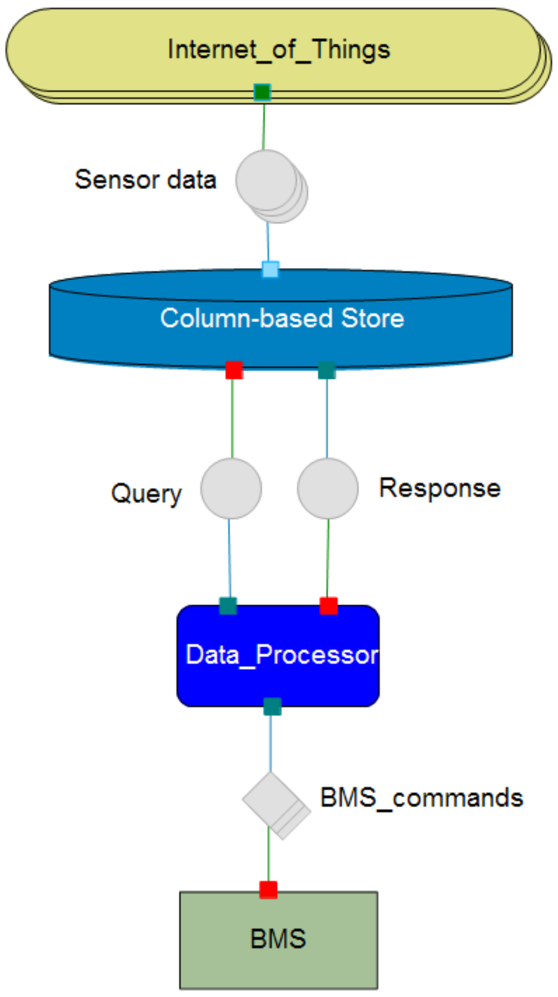


Figure 5.3: System Architecture with Column-Based Store and Distributed Data Processing

5.1.3 System with Column-Store and Distributed Data Processing

In Figure 5.3, a high level design of this proposal is visualized. In this architecture, a column-based store will be used as database. In particular, Apache Cassandra is chosen for this analysis. The primary key of the sensor data table consists of six columns. The first one is id of the sensor and it will be used as partition key by Cassandra. This key will be used as input of the consistent hash function used by Cassandra in order to distribute data over the nodes. Rest of the columns of the primary key will be columns related to date of the data and they will be used as clustering key. They are year, month, day, hour and minute columns, defined in this order. These columns will be used the keep data sorted. The data will be sorted by year first. Then, will be sorted by month, then by day and goes on like this. Since Cassandra can handle columns very efficiently and allows more than one columns to be used as clustering key, there is no need to merge date data into one column like we did for DynamoDB.

As data processor, a user written program or a framework can be used. As Apache Spark has built-in integration with Cassandra, it will be used rather than a manually implemented application. Apache Spark runs on a cluster of computers, each of which is a Spark worker. Different from the previous architecture, the work submitter node does not have to retrieve the whole data from the database and distribute it over the cluster. Instead, since Spark supports Cassandra data sources, each Spark worker connects the database and retrieves the data of its own partition and processes them. Thanks to this integration, one can bypass overhead of transmitting the whole data more than one times over the cluster. Each Spark node processes its data and writes the results itself.

This architecture is suitable for a system with moderate number of data sources. Data consumers do not have to be very small since there is a Spark cluster processing the data, which increases the data processing speed significantly. In this architecture, different from Architecture 1.2, data to be processed may be larger. It will not have a negative effect since Spark's Cassandra integration will handle data retrieval and each node will retrieve only the data that it will process. This system is more preferable when a database row contains too much data since Cassandra does not store a row as a whole. Therefore, a row can contain huge amount of data and columns. On the other hand, this system has some maintenance cost since Cassandra is not a provided service like DynamoDB. User has to handle the setup and maintenance.

All of these three architectures are relatively simple, there are only a few elements in the system. Consequently, there are some missing functionalities. The first one is data preprocessing. Often, the incoming data need to be processed somehow, in order to check data validity or transform the data to some other form. In order to achieve this functionality, there has to be additional elements in the system. The second functionality is there is no mechanism for fast retrieval of the data of the recent past. This functionality is needed since there is often need to process recent data in order to take rapid action for changing conditions. Querying the core database may be unwanted because of speed and consistency reasons. For these purposes, additional architectures will be proposed in following architectures.

5.2 Architecture with NoSQL Database, In-Memory Database and Data Processor

In this architecture set, solutions for fast and consistent retrieval of recent data will be proposed. As stated before, this is needed in order to process recent data to take rapid actions according to the changes in incoming data. Three alternatives will be proposed in this section. There is no major change between them, only specific components will change.

Improvements in this set of architectures are mainly for the 'Step 2-b, watching building demand' operation defined in the reference energy optimization system.

5.2.1 System with Key-Value Store, In-Memory Database and Manual Data Processing

In Figure 5.4, a high level design of this proposal is visualized. In this architecture, a key-value store is used as database, specifically Amazon's DynamoDB. Primary key of the sensor data table is chosen as hash and range type. A unique sensor id is used as hash key and timestamp is used as range key. In order to keep timestamp, a string value created from concatenation of year, month, day, hour and minute values. In order to be able to sort data by date, concatenation is done in stated order.

In addition to the main database, there is another key-value store in this architecture. Its key property is it keeps the data in memory, not on disk. Any key-value store that supports in-memory storage can be used here but Redis is proposed. It also has mechanism for data expiration, which perfectly fits purpose of this database in the system. The key role of this database is to hold recent data flowing from sensors. Data are

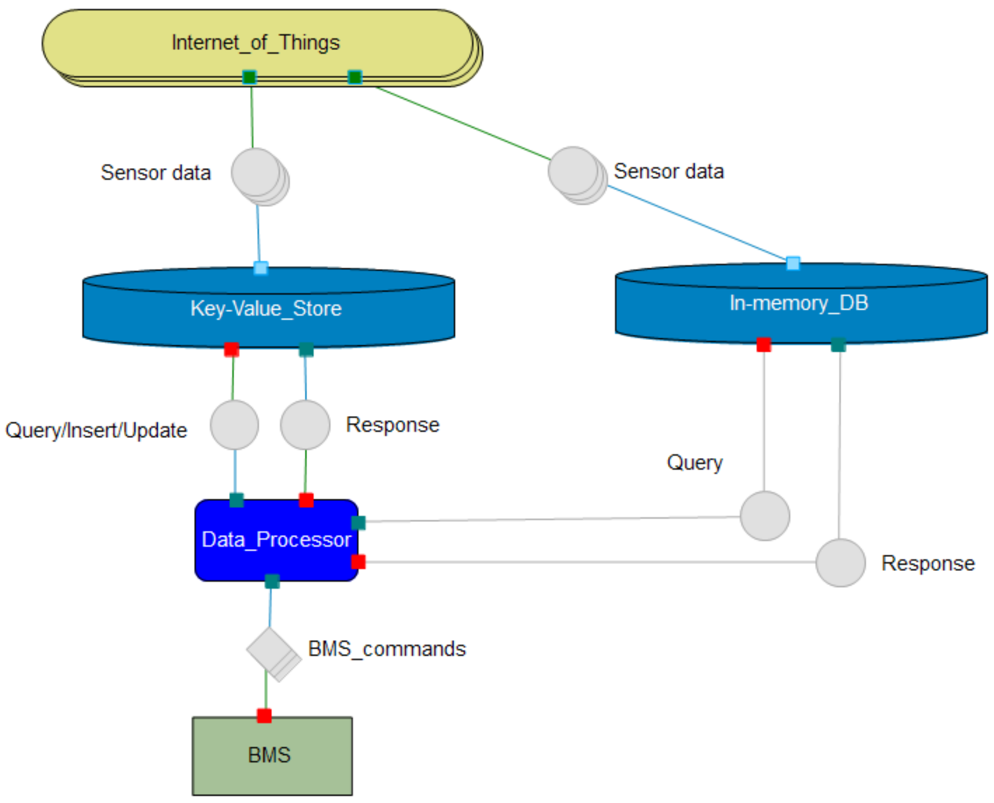


Figure 5.4: System Architecture with Key-Value Store, In-Memory Database and Manual Data Processing

written to main database and this one in parallel. But the data in this database will expire after a pre-defined time. This time is chosen according to the oldest data needed while performing near-real-time processing on recent data. If selected database provides this feature, data will be inserted with expiration time. Otherwise, user has to implement a mechanism to remove data from the database when it expires.

A data processor application retrieves data from both databases. If it is performing a long-term analysis, the main database will be used. While performing recurring short-term analysis, the in-memory database will be used. After analysis is performed, the result will be stored in the main database and appropriate actions will be taken on automation system. No data is written to the in-memory database since the data in this database are volatile. In this architecture, data processing application is written manually.

This architecture is suitable for a system with large number of data sources and moderate number consumers or development purposes. As consumer number increases too much, this system may suffer from performance problems. Moreover, this system cannot handle queries that run over very large data sets and return large amount of data. In addition, data processing performance of this system is not high since processing is performed on a single application. This system is not suitable when a database row contains too much data because DynamoDB has a limit on the size of the data that can be stored in a row. This system has small amount maintenance cost. Moreover, this architecture is more useful when there is a need for continuous processing of small data sets thanks to the in-memory database.

5.2.2 System with Key-Value Store, In-Memory Database and Distributed Data Processing

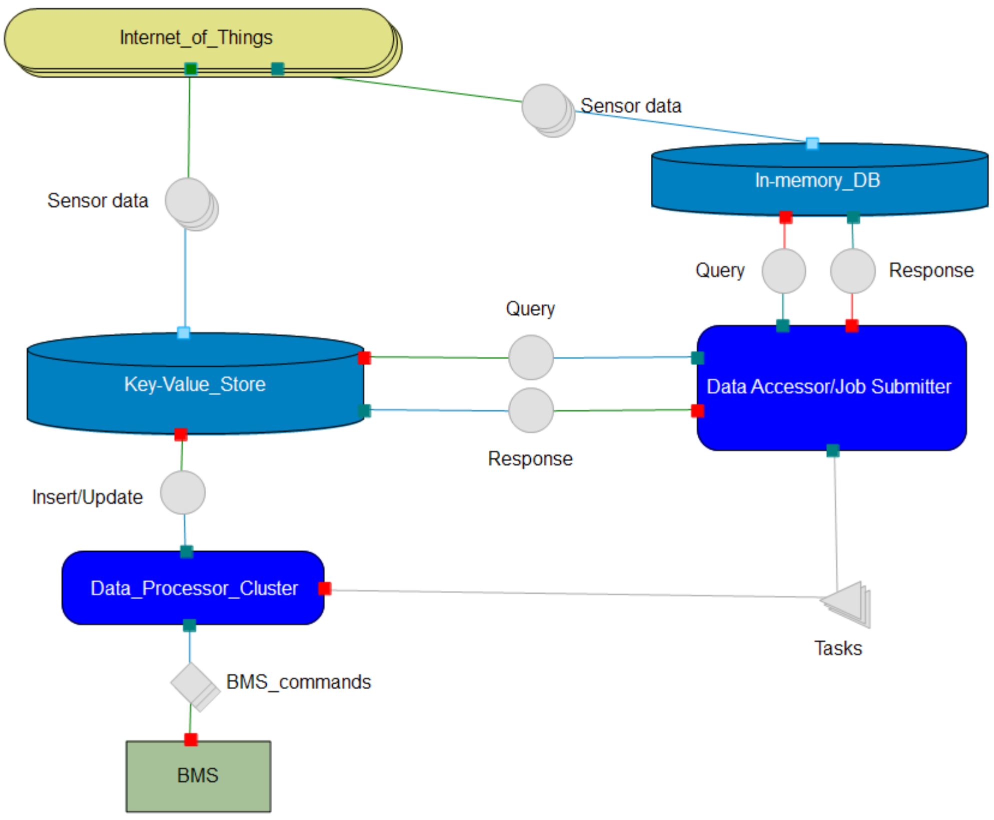


Figure 5.5: System Architecture with Key-Value Store, In-Memory Database and Distributed Data Processing

In Figure 5.5, a high level design of this proposal is visualized. In this architecture, the database part is the same as previous architecture. Amazon's DynamoDB is used as database. The primary key of the table is hash and range type. A unique sensor id is used as hash key in order to store data of same sensor together. As the range part of the key, timestamp is used. In order to keep timestamp, a string value created from concatenation of year, month, day, hour and minute values. Concatenation of the values is done in stated order, in order to be able to sort data by date.

In addition to the main database, there is a parallel key-value store in this architecture. Its key property is it keeps the data in memory, not on disk and enables user to access recent data quickly. Any key-value store that supports in-memory storage can be used here but Redis is proposed. It also has mechanism for data expiration which perfectly fits purpose of this database in the system. The key role of this database is to hold recent data flowing from sensors. Data are written to main database and the in-memory database at the same time. But the data in this database will expire after a pre-defined time. This time is chosen according to the oldest data needed while performing near-real-time processing on recent data, in other words the oldest data needed for 'Step 2-b, watching the building demand' operation. If selected database provides this feature, data will be inserted with expiration time. Otherwise, an expiration mechanism has to be implemented to remove data from the database when they are no longer needed.

Data processor is not a user written application in this system. A MapReduce framework will be used. As proposed in other architectures, Apache Spark will be used. Data processor is not a single node. It is a cluster of computers that run as Spark nodes. However, still there has to be a node that submits works to the Spark cluster. Apache Spark does not have built-in integration or library for DynamoDB integration since DynamoDB is proprietary service of Amazon and Spark is open source. As a drawback of this architecture, the work submitter node has to first get the information data from DynamoDB. Then, it has to distribute the data over the Spark cluster. This can be achieved by 'parallelize' method in Spark. With this method call, any local data can be distributed over the cluster. However, if the required data amount is too large, this operation may be slow since there will be two time transmission of the whole data. First, the whole data are retrieved from DynamoDB, then the whole data are transmitted to the nodes. The data distribution from the in-memory database will not be a problem since the data amount is not very large there and data is retrieved from in-memory database only when required data amount is small. After finishing the data processing, the processor writes the result to the main database since data in in-memory database is volatile and stored as an input for processing of recent data.

This architecture is suitable for a system with large number of data sources and moderate number consumers. As consumer number increases too much, this system may suffer from performance problems. Moreover, this system cannot handle queries that run over very large data sets and return large amount of data. On the other hand, data processing performance of this system is better than the previous architecture since processing is performed by a Spark cluster. This system is not suitable when a database row contains too much data because DynamoDB has an upper limit on the size of the rows. This system has small amount of maintenance cost. This architecture is more useful when there is a need for continuous processing of small data sets in addition to the distributed processing power.

For continuous data processing from in-memory database, a small user written application may be used in order to bypass overhead of network communication while using Spark cluster. This method will be useful when data amount in in-memory database is not very large. However, if data amount retrieved and processed from the in-memory database gets larger as the system gets larger, the single application system may suffer from performance problems. At that point, continuous processing tasks should be passed to Spark cluster.

5.2.3 System with Column-Store, In-Memory Database and Distributed Data Processing

In Figure 5.6, a high level design of this proposal is visualized. In this architecture, a column-based store will be used as database. In particular, Apache Cassandra is chosen for this particular analysis. The primary key of the sensor data table consists of six columns. The first one is id of the sensor and it will be used as partition key by Cassandra. This key will be used as input of the consistent hash function used by Cassandra in order to distribute data over the cluster. Rest of the columns of

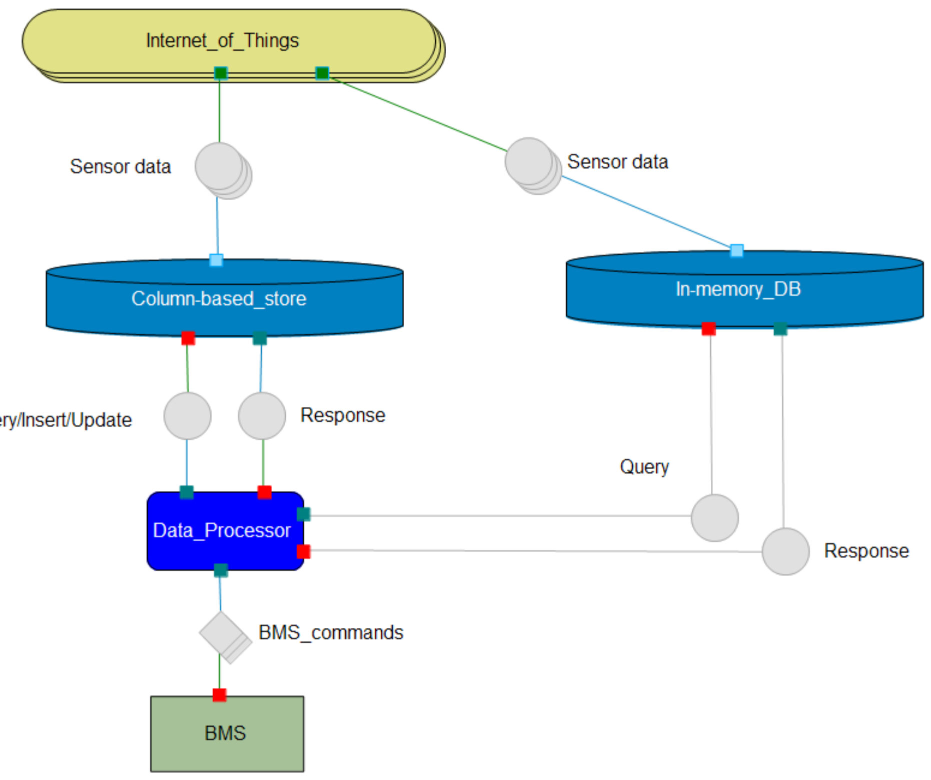


Figure 5.6: System Architecture with Column-Based Store, In-Memory Database and Distributed Data Processing

the primary key will be columns related to date of the data and they will be used as clustering key, namely for keeping data sorted. They are year, month, day, hour and minute columns, defined in this order. These columns will be used to keep data sorted. The data will be sorted by year first. Then, will be sorted by month, then by day and goes on like this. Since Cassandra can handle columns very efficiently and allows more than one columns to be used as clustering key, there is no need to merge date data into one column like we did in DynamoDB.

In addition to the main database, there is another key-value store in this architecture. Its key property is it keeps the data in memory, not on disk. Any key-value store that supports in-memory storage can be used here like Section.5.2.1 and Section.5.2.2. It also has mechanism for data expiration, which perfectly fits purpose of this database in the system. The key role of this database is to hold recent data flowing from sensors. Data are written to main database and the in-memory database in parallel. But the data in this database will expire after a pre-defined time. This time is chosen according to the oldest data needed while performing near-real-time processing on recent data. Among the operations defined for HVAC EOS, mostly the 'Step 2-b, watching building demand' and 'Step 2-c, alarms & notifications' operations require near-real time processing. If selected database provides expiration time feature, data will be inserted with expiration time. Otherwise, user has to implement a mechanism to remove data from the database when they expire.

As data processor, a user written program or a framework can be used. As Apache Spark has built-in integration with Cassandra, it will be used rather than a user written application. Apache Spark runs on a cluster of computers, each of which is a Spark worker. Different from the previous architecture, the work submitter node does not have to retrieve the whole data from the database and distribute it over the cluster. Instead, since Spark supports Cassandra to be used as data source, each Spark worker connects the database and retrieves the data of its own partition and processes them. Thanks to this integration, one can bypass overhead of transmitting the whole data more than one times over the cluster, which is present in architectures that used database does not have integration with Spark. Each Spark node processes its data and writes the results itself. However, if Spark does not have built-in integration with the in-memory database, the work submitter node may have to retrieve data from in-memory database and distribute it over the cluster. Nonetheless, the data distribution from the in-memory database will not be a problem since the data amount is not very large and data are retrieved from in-memory database only when required data amount is small. After finishing the data processing, the processor writes the result to the main database since data in in-memory database is volatile and stores only input data for processing of recent data.

For continuous data processing from in-memory database, a small user written application may be used in order to bypass overhead of network communication while using Spark cluster. This method will be useful when data amount in in-memory database is not very large. However, if data amount retrieved and processed from the in-memory database gets larger as the system gets larger, the single application system may suffer from performance problems. At that point, continuous processing tasks should be passed to Spark cluster.

This architecture is suitable for a system with moderate number of data sources. Data consumers does not have to be very small since there is a Spark cluster processing the data, which increases the data processing speed significantly. In this architecture, different from the previous one, data to be processed may be larger. It will not have a negative effect since Spark's Cassandra integration will handle data retrieval and each node will retrieve only the data that it will process. This system is more preferable when a database row contains too much data since Cassandra does not store a row as a whole. Therefore, a row can be enormous. On the other hand, this system has moderate amount of maintenance cost since Cassandra is not an offered service like DynamoDB. User has to handle the setup and maintenance. In addition, thanks to the in-memory database, this architecture is more preferable when there is a need for continuous processing of small data sets.

These three architectures does not bring major changes or upgrades to the ones in Section.5.2. Instead, they focus on performance of the continuous processing needs of systems. They aim to solve quick processing of data when there is no data-stream processing feature in the system. This features enables quick processing of incoming data for 'watching building demand' or 'alarms & notifications module'. These architectures are not suitable for very large systems. When data volume and velocity gets too large, there will be performance problems while processing large amounts of data from the in-memory database. After that point, parallel processing power of Spark cluster will outperform single application processing of in-memory database and in-memory database can be removed from the system.

5.3 Architecture with Data Warehouse

This set of architectures focuses on a system that stores very large data and also queries large amounts of data. Moreover, solution proposals in this section will add SQL-like query capabilities to NoSQL systems. For example, most NoSQL systems does not support join operation. Using a data warehouse software, executing this kind of queries will be possible and user will be able to do database operations by writing SQL-like commands instead of writing different codes for different NoSQL systems.

There are four alternatives in this section. Three of them are experimental since the framework support of those alternatives are still under development. All of these proposals use Apache Hive as data warehouse that handles data access and/or processing. As mentioned in Section.2.4.3, Hive is not suitable for user interactions and operations that need rapid response. Therefore, in addition to Hive cluster described in this section, a parallel data processing solution can be run as described in Section.5.2 for operations with smaller data set and require faster response.

5.3.1 System with Key-Value Store and Data Warehouse on Hadoop Cluster

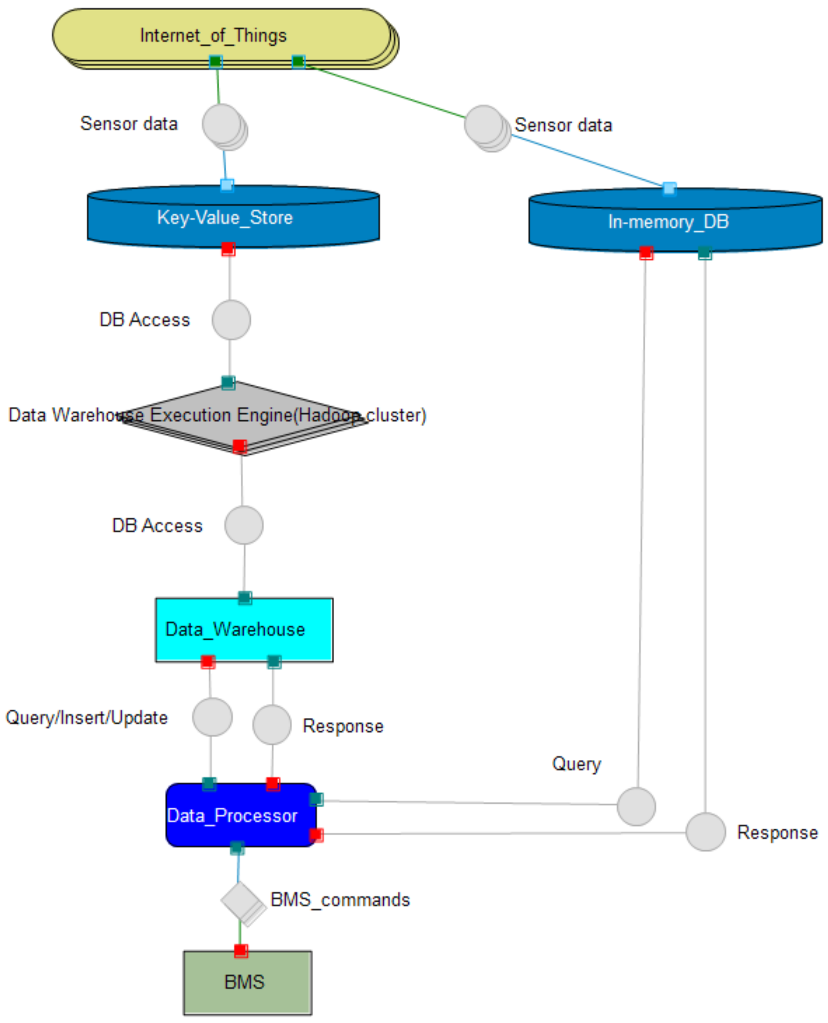


Figure 5.7: System Architecture with Key-Value Store and Data Warehouse on Hadoop Cluster

In Figure 5.7, a high level design of this proposal is visualized. In this architecture, as database, a key-value store is used. As proposed before, the system will be described with Amazon's DynamoDB. As primary key of the sensor data table, hash and range type of primary key is used. A unique sensor id is used as hash key in order to store data of same sensor together. As the range part of the key, timestamp is used. In order to keep timestamp, a string value created from concatenation of year, month, day, hour and minute values. In order to be able to sort data by date, concatenation is done in stated order.

In addition to the main database, another key-value store may be used. Its key property is it keeps the data in memory, not on disk. Any key-value store that supports in-memory storage can be used here but Redis is proposed. This part of the architecture is same as the ones explained in Section.5.2. However, since data size is assumed to be very large in this solution, this additional database may not be very suitable. The proposed data warehouse framework may perform better.

Data processor nodes do not have direct access to the database. Instead, there is a data warehouse cluster in-between. Apache Hive is a de-facto software for data warehousing on Hadoop, so solution will be described over it. Apache Hive allows users to write SQL-like queries. It converts them to appropriate form for the database behind Hive. Moreover, as default execution engine, it runs on a Hadoop cluster. Therefore, it allows MapReduce style processing on very large data sets during queries. In addition, it can perform complex query operations like join or filter very efficiently.

A data processor application or software will be in communication with Hive and read/write data through it. Spark is default data processor of this solution proposal since it supports use of Apache Hive as data source. If an in-memory database is used, the data processor connects to it directly. Using Hadoop and HDFS for small amounts of data will not be very efficient because of overhead of Hadoop and HDFS. They are useful when data cannot be processed in traditional single-application methods. After analysis is performed, the result will be stored in the main database through Hive and appropriate actions will be taken on automation system.

This architecture is suitable for a system with large number of data sources and consumers. Apache Hive can handle large number of concurrent access and query processing. Moreover, this system can handle queries that run over very large data sets and return large amount of data. In addition, data processing performance of this system is high since processing is performed on a Spark cluster. This system is not suitable when a database row contains too much data because DynamoDB has a limit on the size of the data that can be stored in a row. This system has high maintenance cost since there are different clusters that need to be managed. This system should be used when there is complex query needs on very large data sets. If queried data set is not very large, the overhead of Hive will cause performance degradation rather than an improvement.

5.3.2 System with Column-Store and Data Warehouse on Hadoop Cluster

In Figure 5.8, a high level design of this proposal is visualized. In this architecture, a column-based store will be used as database. In particular, Apache Cassandra is chosen for specific analysis. The primary key of the sensor data table consists of six columns. The first one is id of the sensor and it will be used as partition key by Cassandra. This key will be used as input of the consistent hash function used by Cassandra in order to distribute data over the nodes. Rest of the columns of the primary key is related to date of the data and they will be used as clustering key. They are year, month, day, hour and minute columns, defined in this order. These columns will be used the keep data sorted. The data will be sorted by year first, then month and goes on. Since Cassandra can handle columns very efficiently and allows more than one columns to be used as clustering key, there is no need to merge date data into one column. As proposed in previous section, in addition to the main database, an in-memory database may be used depending on the data volume and rate and user choice.

Data processor nodes do not have direct access to the database. Instead, there is a data warehouse cluster in-between. Apache Hive is de-facto software for data warehousing. Thus, solution will be described over it. Apache Hive allows users to write SQL-like queries. It converts them to appropriate form for the database behind Hive. Moreover, as default execution engine, it runs on a Hadoop cluster. Therefore, it allows MapReduce style processing on very large data sets during queries. In addition,

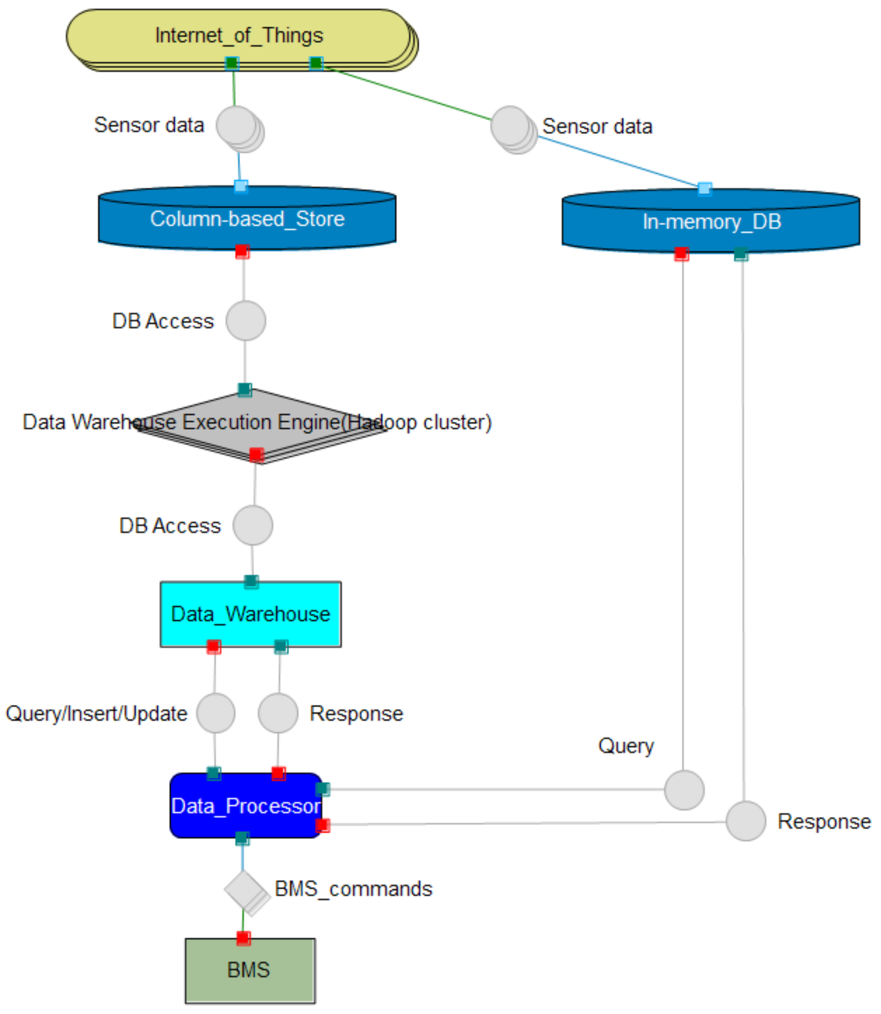


Figure 5.8: System Architecture with Column-Based Store and Data Warehouse on Hadoop Cluster

it can perform complex query operations like join or filter very efficiently.

A data processor application or software will be in communication with Hive and read/write data through it. Since Spark supports use of Apache Hive as data source, Spark is default data processor of this solution proposal. If an in-memory database is used, the data processor connects to it directly, because that database is used for retrieving small amount of data. Using Hadoop and HDFS for small amounts of data will not be very efficient because of their overhead. They are useful when data cannot be processed in traditional single-application methods. After analysis is performed, the result will be stored in the main database through Hive and appropriate actions will be taken on automation system.

This architecture is suitable for a system with large number of data sources. Data consumer amount can be high since there is a Spark cluster processing the data and it retrieves data through Hive. This system can handle complex queries that run over very large data sets and return large amount of data thanks to Apache Hive. This system is more preferable when a database row contains too much data since Cassandra does not store a row as a whole and does not limit row size. On the other hand, this system has high maintenance cost since Cassandra is an offered service like DynamoDB and there other clusters to manage like Apache Hive and Spark. User has to handle the setup and maintenance. This system should be used when there is complex query needs on very large data sets. If queried data set is not very large, the overhead of Hive will cause performance degradation rather than an improvement.

Since there is no official support for Hive and Cassandra integration, this architecture is a theoretical proposal. There are a few libraries for the integration but they are not stable and officially supported yet.

5.3.3 System with Key-Value Store and Data Warehouse on Spark Cluster

In Figure 5.9, a high level design of this proposal is visualized. In this architecture, as database, a key-value store is used. As proposed before, the system will be described with Amazon's DynamoDB. Table properties are the same as in Section.5.3.1. An additional in-memory database can be used for fast retrieval of small amount of recent data as described in Section.5.3.1.

Data processor nodes do not have direct access to the database. Instead, there is a data warehouse cluster in-between. Apache Hive is a de-facto software for data warehousing. Thus, solution will be described using it. Apache Hive allows users to write SQL-like queries. It converts them to appropriate form for the database behind Hive. In this architecture, different from the architecture in Section.5.3.1, Hive will be running on a Spark cluster as its execution engine. Therefore, it allows queries that return very large data sets. If the queried data fit in total memory of the Spark cluster, query will be processed much faster than Hadoop execution engine. In addition, by

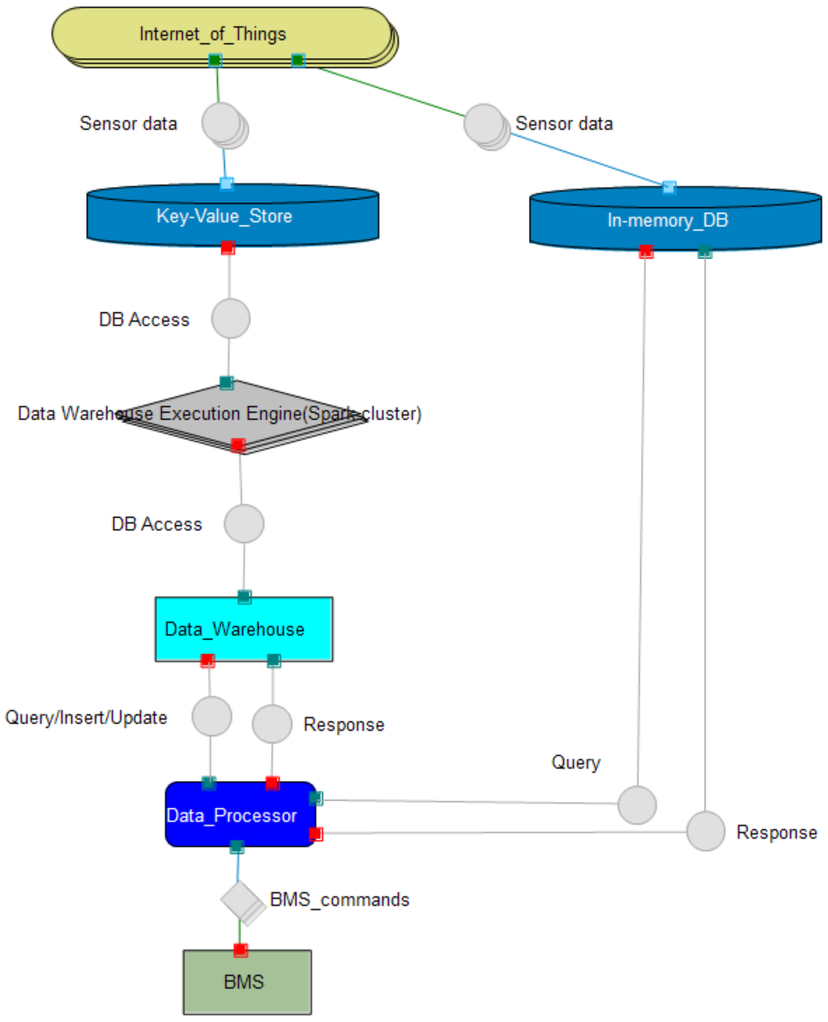


Figure 5.9: System Architecture with Key-Value Store and Data Warehouse on Spark Cluster

making use of MapReduce style processing of data, it can perform complex query operations like join or filter very efficiently.

A data processor application or software will be in communication with Hive and read/write data through it. Since Spark supports use of Apache Hive as data source, Spark is default data processor of this solution proposal. If there is an in-memory database in the system, the data processor connects to it directly. After analysis is performed, the result will be stored in the main database through Hive and appropriate actions will be taken on automation system.

This architecture is suitable for a system with large number of data sources and consumers. Apache Hive can handle large number of concurrent access and query processing. Moreover, this system can handle queries that run over very large data sets and return large amount of data. In addition, data processing performance of this system is high since processing is performed on a Spark cluster. This system is not suitable when a database row contains too much data because DynamoDB has an upper limit on row size. This system should be used when there is complex query needs on very large data sets. If queried data set is not very large, the overhead of Hive will cause performance degradation rather than an improvement. Since the Hive's execution engine is Spark in this architecture, this system will perform when queried data set is very large, but also small enough to fit in total main memory of the Spark cluster.

This architecture is a theoretical proposal at the moment since the Spark execution engine support for Apache Hive is still under development. It is developed by official developers, so it will be supported completely and ready for tests in near future.

5.3.4 System with Column-Store and Data Warehouse on Spark Cluster

In Figure 5.10, a high level design of this proposal is visualized. In this architecture, a column-based store will be used as database. In particular, Apache Cassandra is chosen for specific analysis. Table properties are the same as in Section,5.3.2. An additional in-memory database can be used for fast retrieval of small amount of recent data as described in Section.5.3.2.

Data processor nodes do not have direct access to database. There is a data warehouse cluster in-between. Apache Hive is going to be used as data warehouse. It allows users to write SQL-like queries. It converts them to appropriate form for the database behind Hive. In this architecture, different from the one in Section.5.3.2, Hive will be running on a Spark cluster as its execution engine. Therefore, it allows queries that return very large data sets. If the data that will be queried fits in total memory of the Spark cluster, query will be processed much faster than Hadoop execution engine. In addition, by making use of MapReduce style processing of data, it can perform complex query operations like join or filter very efficiently.

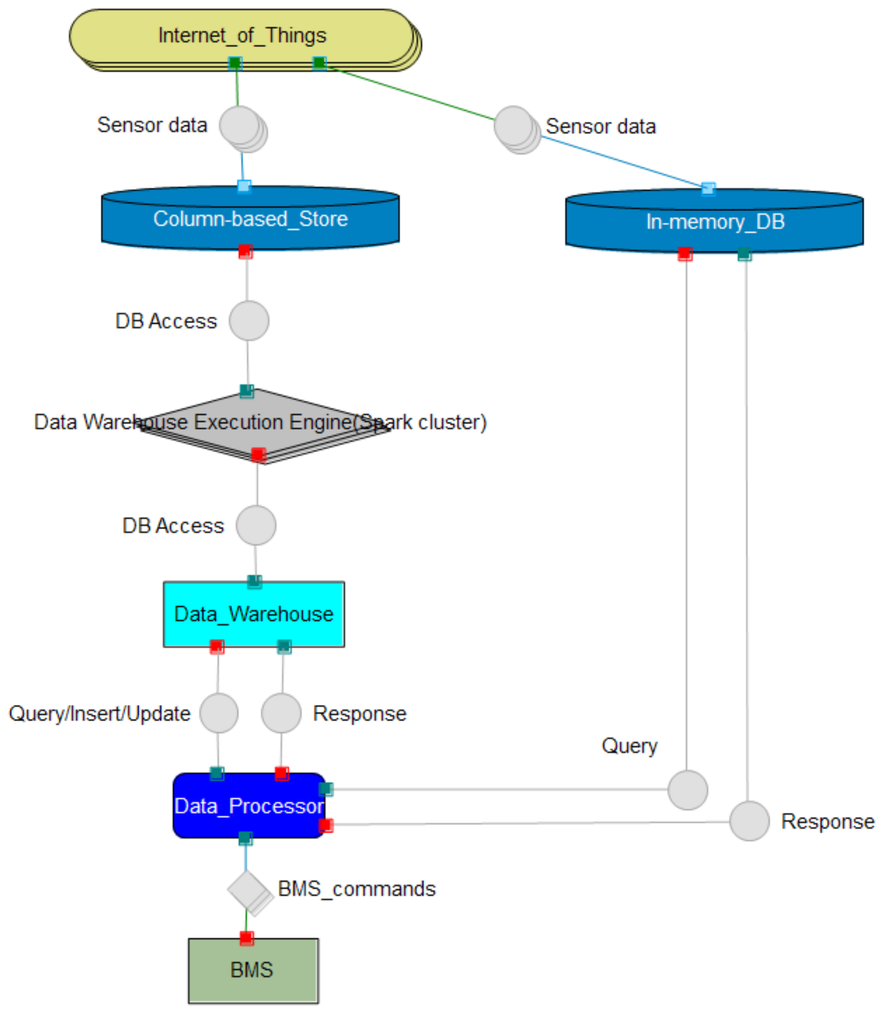


Figure 5.10: System Architecture with Column-Based Store and Data Warehouse on Spark Cluster

A data processor application or software will be in communication with Hive and read/write data through it. Since Spark supports use of Apache Hive as data source, Spark is default data processor of this solution proposal. If there is an in-memory database in the system, the data processor connects to it directly. The result will be stored in the main database through Hive and appropriate actions will be taken on automation system after analysis is performed on by Hive.

This architecture is suitable for a system with large of data sources. Data consumer amount also can be high since there is a Spark cluster processing the data and it retrieves data through Hive. This system can handle complex queries that run over very large data sets and return large amount of data thanks to Apache Hive. This system is more preferable when a database row contains too much data since Cassandra does not limit row size. On the other hand, this system has high maintenance cost since there are several clusters to manage. This system should be used when there is complex query needs on very large data sets. If queried data set is not very large, the overhead of Hive will cause performance degradation rather than improvement. Since the Hive's execution engine is Spark in this architecture, this system will perform when queried data set is very large, but also small enough to fit in total main memory of the Spark cluster.

This architecture is also a theoretical proposal at the moment since the Spark execution engine support for Apache Hive is still under development. Since it is developed by official developers, it will be supported completely and ready for tests in near future.

This set of architectures focuses on performance of large queries on very large data sets. Queries like this are used in operations like forecasting the future energy consumption based on past data, as described in Section.4.3.2.4. In the example EOS, at most data for one year is needed. However, the required data amount may change depending on the algorithm or analysis. Data from multiple buildings or a whole region may be needed in the future for different analyzes. Therefore, the system has to be able to handle very large queries and process them.

5.4 Architecture with Data Stream Handler

In Figure 5.11 a high level design of this proposal is visualized. This architecture focuses on the processing of data during transmission before it is written to the database. In all of the previous architectures, number of data sources was not very high and data were written to the database directly without any processing performed. If data were processed, it would bring overhead to the node that retrieves and writes the data and some of the data might have been lost while waiting for node to process previous data. The software architecture after the database can be like any of the previous architectures. Nothing new will be introduced in that part. However, the suggested one was

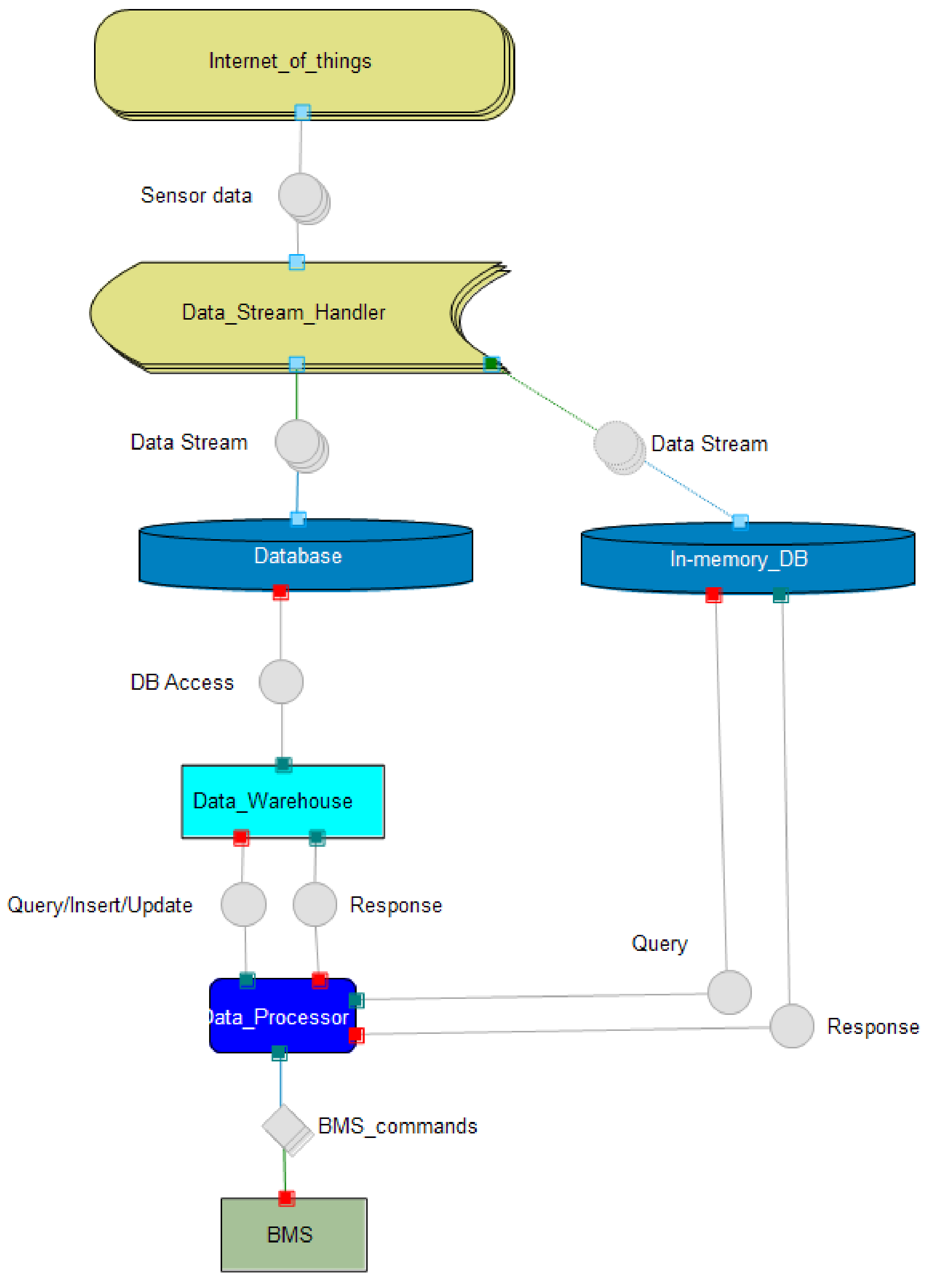


Figure 5.11: System Architecture with Data Stream Handler

shown in the diagram.

The incoming data are first written to the Amazon's Kinesis service. Kinesis can continuously capture and store terabytes of data per hour from hundreds of thousands of sources. Then, a data preprocessor application, a single application or multiple ones written by the user, retrieves data from this service and applies transformation to data and processes them. Then, data are written to the database. If there is a need to take quick action on incoming data, the preprocessor can access to the automation system. The processing needs that are not very urgent were left to the data processor, which processes data in the database, in order not to slow down data preprocessing pipeline. At some point, if incoming data velocity exceeds the preprocessing velocity of the preprocessor node, data will not be lost. Data will wait in Kinesis stream until they are retrieved. The preprocessor node will consume the remaining data when its work load gets lighter.

This architecture can be applied to any architecture defined in sections 5.1, 5.2 or 5.3. This system allows preprocessing of incoming data up to some point without any data loss. If velocity of incoming data exceeds the preprocessing speed and remains constant, or all of the data analysis needs to be done in near-real time before writing data to the database, then the next architecture proposal should be used for better stream processing speed.

This data-stream processing solution can replace the in-memory database solution if it can hold small amount data. This feature depends on the used framework. Moreover, another important usage of this solution will be in ‘Step 2-c, alarms & notifications’ operations. A user-defined alarm can be triggered immediately before writing data to any database. However, if there are a lot of preprocessing operations in multiple steps, this system may suffer from performance problems when data velocity increases. A solution to this problem is using a distributed data processing pipeline, which is defined in Section.5.5.

5.5 Architecture with Data Stream Handler and Distributed Data Stream Processor

In Figure 5.12, a high level design of this proposal is visualized. This architecture focuses on the processing and transmission of data before they are written to the database like in Section.5.4. Different from architecture, data stream will be processed on a real-time distributed stream processing cluster. The architecture after the database can be any of the previous architectures. Nothing new will be introduced in that part. However, the suggested one was shown in the diagram.

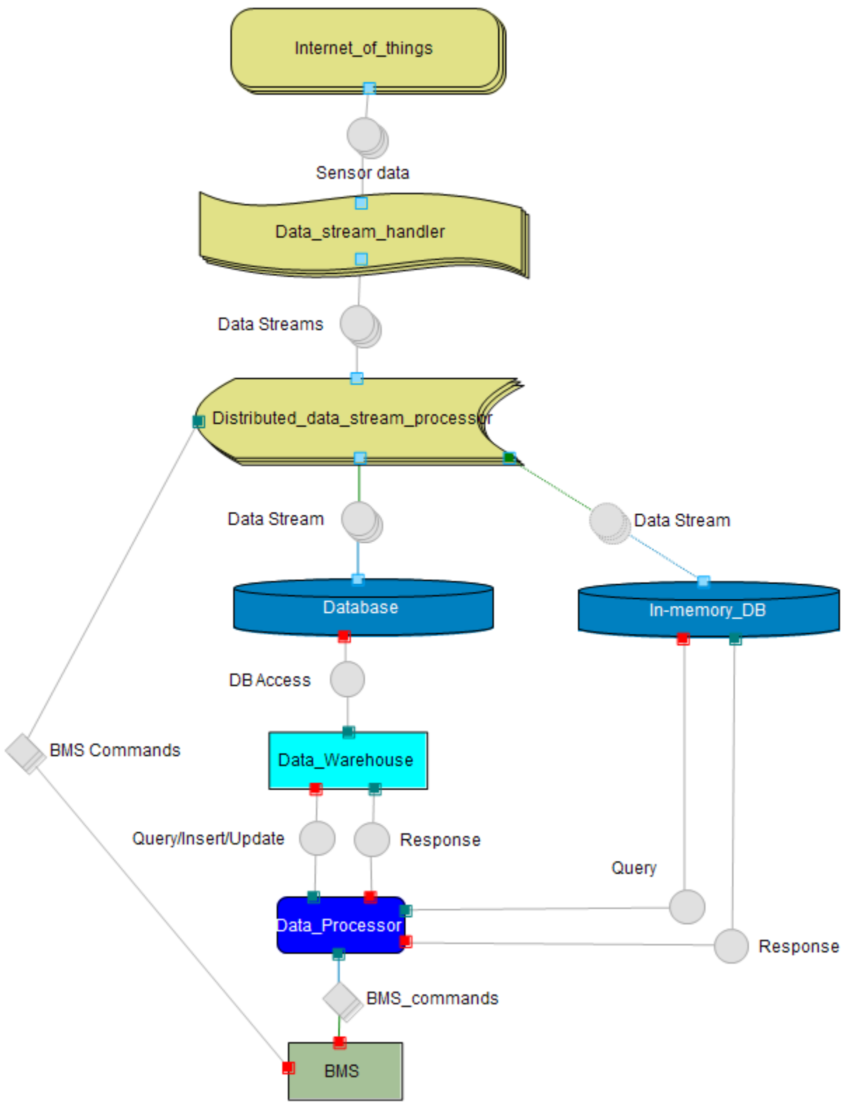


Figure 5.12: System Architecture with Data Stream Handler and Distributed Data Stream Processor

The incoming data are first written to the Amazon's Kinesis service. Kinesis can continuously capture and store terabytes of data per hour from hundreds of thousands of sources. Different from the architecture in Section.5.4, data are processed by a distributed stream processing cluster instead of a single application. The most common framework is Apache Storm and fits perfectly for applications that perform multiple transformations and processing on incoming data. Storms data source, called spot in Storm terminology, is Amazon Kinesis. Then, for each step of transformation or processing, data are forwarded to the next worker node, called bolt in Storm terminology. At the end of the Storm pipeline, data are written to the database. Since Storm works on a cluster, its processing power is a lot more than a single application. Moreover, Storm scales very easy and good by adding new nodes to the cluster. Thanks to these two properties of Storm, all the processing and data checks that needed for a single data unit can be performed in Storm's pipeline, by assigning different operations on each bolt. The data processor only performs analysis that requires large time-series data sets. An example architectural diagram that shows how Kinesis and Storm works is given in Figure 5.13

Apache Storm is used for data stream processing purpose. Data stream processing enables data to be processed before they are written to disk. If this method was not used, for each analysis, database would need to be queried. Thanks to this method, analyzes and operations that do not rely on data from different sources can be performed during the flow of the data from sensor to database. By this way, a huge burden is taken away from the database. In a data stream processing cluster, very simple operations are performed on each node and then data are forwarded to the next node. Here is not the right place for complex processing. Therefore, performance of a data stream processing cluster mostly depends on the configuration of the cluster and communication delay between the nodes. After data are generated, they are immediately ready to be processed by data stream processing cluster. This processing method is not about processing of large data sets, it is about processing of single data units.

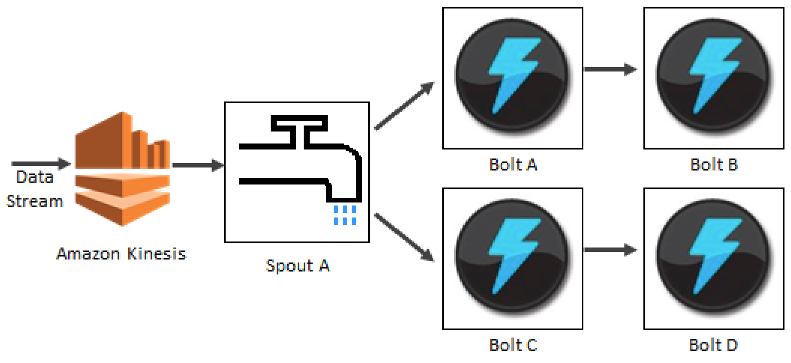


Figure 5.13: Amazon Kinesis and Apache Storm Working Together [19]

This architecture can be applied to any of the systems defined in sections 5.1, 5.2 or 5.3. Data processing speed is not limited like in Section.5.4 since stream-processing component is scalable in this one. If data processing suffers from performance issues, another node can be added to the cluster easily.

As an improvement to the architecture in Section.5.4, this data streaming and distributed computing solution can handle preprocessing of any amount of data by expanding the cluster. Moreover, multiple steps of data processing and transformation can be defined so that each node performs only one operation.

This architecture can handle any amount of data load. Real-time stream-processing, data storage, and data processing and analyzing components are all scalable ones. However, this architecture needs several clusters. Therefore, initial hardware and maintenance costs will be more than all of the previous architectures and will get higher as clusters expand.

CHAPTER 6

IMPLEMENTATION & TESTING

A typical IoT system has five main components. The first one is the network of things, which is the data source of the system. IoT can be described as a cloud of sensors, which generates data continuously. The second component is data storage components. This is usually a NoSQL data store in BigData environments. Next component is the data processor. Data processor uses the stored data to do technical analyzes and sends commands or data to other two components. The fourth component is data visualization component. It is used to visualize incoming data in a human-understandable format. The last component is the management system, which manages a sensor network, for example a building management system (BMS). These components and communication and processing mechanisms between them are described in detail in Chapter 5.

In order to design and implement such a system, we have proposed software architectures. Now, we are going to test internals of those software architecture proposals. Using the results of these tests, we will be analyzing the architectures and tools together first to decide which architecture to select and then to decide which tool suits better for our recommended architecture proposal.

Before this study, there were several studies that test performance of BigData and NoSQL systems, for example the study in [31] and study in [39]. In study [39], researchers performed tests with 120 million records and each record was 1 KB. During the tests they performed only basic read/write/update operations. Narde conducted tests with 50 million records and again each record was 1 KB [31]. The database performances were tested and compared in terms of execution time of large amount of queries. In both studies, researchers just perform basic operations and they do not express any query, table configurations and index details. Since we performed some table and index configuration for our data format in previous chapter, these tests are not enough for us. Moreover we do not know what type of queries they had executed. Our queries are mostly interval based queries rather than simple 1-record insert/get operations of previous tests. Therefore, we had to perform our tests with our configurations and settings. Different from previous ones, our tests define data format, table configuration and indexing in detail and give query details for the tests.

Besides the database subsystem, there is also a stream processing sub-system in our proposed software architecture. About that subsystem, there is no specific condition in our system. Stream processing frameworks do not care about the data format. They just propagate data through nodes and execute given operation. In stream processing, operations are very simple, so stream processors do not focus on processing power. Therefore, performance of stream processing subsystem is about the system configuration and the communication delay between nodes.

6.1 Components of Proposed Architectures to Test

Several architectures with different tools and structures were proposed. They have mostly common components but in some of them there are additional ones. We will pass over these components and clarify relations between different architecture proposals.

Architecture with NoSQL Database and Data Processor: This is the most basic architecture. It is described in detail in Section.5.1. There are two main components. First one is the NoSQL database. Data from the IoT are collected and stored here. The other component is data processor. This can be a single application or a distributed data processing framework. Decision of data processor depends on the system load.
The data storage component of this architecture is present in all of the architectures and it is the core of the system since it handles the storage of the whole data.

Architecture with NoSQL Database, In-Memory Database and Data Processor: This architecture is very similar to the first one, except for an additional component. It is described in detail in Section.5.2. There is an in-memory database that runs in parallel with the core NoSQL database. Data from the IoT flow to both of them at the same time but in-memory one holds the data for very short time in order to access recent data quickly.

Again in this architecture, NoSQL database is the common component among the alternatives and it is the main database. Data processor component accesses to both NoSQL and in-memory database, if present, at the same time and uses in-memory one for only short-term processing.

Architecture with Data Warehouse: Again, this architecture improves the previous two ones by adding one more BigData component. It is described in detail in Section.5.3. All of the components in previous architectures are present in this one as well. There is a data warehouse as the new component and it stands between the NoSQL database and data processor node. This component adds capability of Hadoop based map-reduce style processing of data during the query execution. This architecture is mostly suitable when queries contain enormous amount of data since map-reduce paradigm has its own overhead and is not efficient for small data-sets.

Other components of the system (NoSQL database, data processor, in-memory database) are the same as previous architectures.

Architecture Data Stream Handler: This architecture deals with the flow of the data towards the system. It is described in detail in Section.5.4. The data from the IoT enter to data stream handler. This component controls data flow with very high volume and rate. It can capture high volume of data before the system so that no data packet is lost or unprocessed because of the non-homogenous and high-rated data flow. This component delivers data to the NoSQL database (and in-memory database if present). After the NoSQL database, rest of the system is the same as previous architectures.

Architecture Distributed Data Stream Processor: This architecture aims processing of the data on the fly, right after it is generated. It is described in detail in Section.5.5. Again, overall system architecture does not change very much.

There is a new component between the data stream handler and the database. This components is called distributed data stream processor. Data are delivered to this component instead of the database. Stream processor processes each data unit and extracts data from them. Then, it decides either to send the data to the database or discard them. Since this component can run in a distributed environment, it can handle very large data volume and rate. After data leave the stream processor, rest of the system is one of the first three architectures.

6.2 Test Scenarios

A subset and combination of the proposed tools will be tested and results will be presented. Testing and environment details will be described. We did not test all of the architectures completely because, without a real up and running system, some of the tests would not be realistic. For example, stream processing tests. Without real hardware and distributed devices and sensors, performing tests in a local test environment would not yield realistic results that help us to evaluate the architecture correctly for real environment. In addition, evaluating stream processing is not simple, because performance is mostly affected by system configuration and communication delay due to the fact that expensive operations are not performed in this component. We also did not test an architecture with data warehouse because our reference system did not require queries with very large data. Since data amount is not that large, results without data warehouse were fast enough and probably faster than data warehouse. Data warehouses has their own map-reduce overhead, so there is no need to introduce this overhead to our fast-enough system. Here are our test scenarios:

a. NoSQL Database tests
   In this set of tests, two NoSQL databases, Cassandra and DynamoDB, are tested for their multiple aspects: ease of installation and maintenance, data access methods they provide and data retrieval speed.

b. Data processing over NoSQL databases with small data sets
   In this set of tests, the same databases are used. The examined feature in this part is, databases' performance for an IoT system like described in Chapter 4 chapter,

rather than just basic query performance. In order to simulate the system, parallel queries are executed on databases.

c. Integration with data processing frameworks
   After performing tests of specific databases, their integration with data processing frameworks are analyzed. Apache Spark is used as the data processing framework.

d. Data processing over NoSQL databases with larger data set
   After performing previous tests, additional tests are performed in order to benchmark data processing performance of selected data processing framework. For this set of tests, firstly analysis is performed without any data processing framework. Data are queried from the database and then mathematical operations are applied. Then, a distributed data processing framework is used for same operation with two different configurations. In the first configuration, the data processor runs on a single node with four threads. In the second configuration, the data processor runs on four different nodes in a distributed environment.

6.3 Test Environments

For testing, two separate environments are used for different tools and purposes.

6.3.1 Amazon Web Services

This environment is built on ‘Amazon Web Services (AWS)’. For DynamoDB, AWS is a must. In addition to NoSQL service of Amazon, Elastic Compute Cloud (EC2) service is used as well. For DynamoDB, users cannot specify technical details. Only requested throughput can be specified. Only one table is used for sensor data. There are two parameters that can be specified by user:

Primary Key: In this table, hash and range type of primary key is used. As hash key, a unique sensor id is used. Thanks to this key, data of same hash key are kept close. As the range key, a timestamp string is used. Timestamp string is created by concatenation of year, month, day, hour and minute values. In order to be able to sort the data correctly by date, concatenation is done in stated order.

Throughput: Since performed tests mostly include read operations write throughput is not important. Read throughput is set to 100 read capacity. This capacity enables user to read 100 items from table every second. In none of the tests, the consumed throughput reached 100. Therefore, we can assume that DynamoDB worked at maximum performance, retrieval speed-wise.

The sensor data table was populated with data of 50 devices. There were two years of data for each device and data are written with frequency of five minutes, meaning for every five minutes, there is only one record for a single device.

Except throughput, another factor that affects data retrieval speed from DynamoDB is network performance of the machine accessing to DynamoDB. Since DynamoDB transfers items in JSON format, even the column names consume network capacity. When performing tests from a local computer, DynamoDB responds fast but data retrieval speed was very slow for large queries. Therefore, the test application was deployed on an Amazon EC2 instance in same region with DynamoDB table. Rest of the speed issues are about Amazon's internal structure and cannot be seen and modified from outside. The test application was deployed on an m3.xlarge type of EC2 instance. Properties of an m3.xlarge instance are:

- Intel Xeon E5-2670 processor with four cores.
- 2 SSD based storage devices, each of which has 40GB capacity.
- 15 GB RAM.
- Network performance classified as 'High'.

The test application has also a web interface. Tests are initiated from the web interface and results are displayed there. Therefore, data transmission is only in Amazon's own system and performance measurements are performed for internal transmission only. Then, only results are returned to the user by the web interface.

6.3.2 Local Cluster

For this environment, a local BigData cluster at the department was used. The cluster consists of four identical computers built for BigData processing. Each of the computers has the following properties:

· 2 x Intel Xeon E5-2630v3 processor. Each processor has eight cores and a computer has sixteen total cores.

· 128 GB memory. (8 x 16 GB).

· 40 Gbps Infiniband network connection.

On this cluster, a Cassandra cluster and a Spark cluster are installed. Only parameters of the tables that store BigData will be explained. Parameters for Cassandra installation:

Primary Key: Primary key of the sensor data table has six columns. First column is the sensor id, which is unique to each sensor. This column is used as hash key like in DynamoDB. It is used to distribute data over the nodes. Cassandra stores data with same hash key together. Therefore, queries that retrieve data of one sensor are executed on a few nodes, rather than all nodes in the cluster. Rest of the columns in the primary key are year, month, day, hour and minute columns. These columns are used as clustering keys in order to store and retrieve the data ordered, like in DynamoDB. However, different from DynamoDB, clustering key does not have to be only one column. Data are sorted by the first clustering column at first. Then, it is sorted by the second clustering column, then third and so on.

Consistency: As described before, consistency is configurable for each query in Cassandra. As default consistency level, 'ONE' is used. 'ONE' consistency level means receiving response from one replica is enough to return response to user.

As described in example scenarios, for each sensor, data come every five minutes. Duration of five minutes is more than enough for Cassandra to write data to every replica. Therefore, there is very tiny chance that user will get out-of-date data or missing data. The worst-case scenario is that user does not get the last inserted data for a sensor upon a request. This may be problem for data preprocessing, watching building demand and alarm and notification-triggering operations as described in Chapter 4, since these operations rely on the most recent data. Using a data stream processing software before writing data to the database can solve this problem. This way, in order to perform those operations, there is no need to query database. Data are analyzed while it is on its way to the database.

When performing forecasting operations (Section.4.3.2.4), user has to query the database for past data. The last inserted data can be missing here if forecasting operation starts before data are propagated to all replicas. However, since there are data of long time periods, only one missing data would not have a dramatic effect on operation result.

Replication Factor: Replication factor is used as two in sensor data tables. This means a data unit is written to two nodes. This way, data are backed up in two nodes and if a node is too busy to return response, other node returns it. Since there are only four computers in the cluster, setting the replication factor too large may cause network congestion and affect the performance negatively.

For tests that connect to Cassandra directly, Java programming language on JVM version 1.7 is used and test applications are run on Linux Ubuntu 14.04 operating system.

The sensor data table on Cassandra was populated with data of 50 devices like just like DynamoDB was populated in Section.6.3.1. There were two years of data for each device and data are written with frequency of five minutes, meaning for every five minutes, there is only one record for a single device.

For Spark cluster installation, no modification is made on Spark's default configuration. Only worker node IP addresses are entered. Spark cluster includes four nodes. During tests, Spark is used in two different modes. In first mode, only one node is used for processing with four threads. In second mode, there is no multi-threading. Four nodes are used for processing in parallel instead of threads. These modes do not rely on cluster configuration. It is specified in test application when submitting the job to tell the cluster how to execute the job. Scala programming language on JVM version 1.7 is used for test applications that run on Spark. In order to use Spark and Cassandra together, additional libraries are used as well. Test applications are run on Linux Ubuntu 14.04 operating system.

6.4 Codes for Querying Databases

In this section, sample generic codes are introduced for querying databases together with the explanation of their parameters. For different queries during tests, parameters will be changed according to the test conditions (mostly date parameters). When presenting test results, specific queries of the test will also be given.

6.4.1 DynamoDB Query Codes

In order to access DynamoDB, Amazon's Java API is used during this study. Here is an example Java code for querying a DynamoDB table:

```
DynamoDB dynamoDb = new DynamoDB(new AmazonDynamoDBClient());
Table table = dynamoDB.getTable(tableName);
QuerySpec querySpec = new QuerySpec()
            .withHashKey("deviceId", deviceId)
            .withConsistentRead(false).withRangeKeyCondition(
              new RangeKeyCondition("timestamp")
                .between(timeStart, timeEnd));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

This code queries a table named 'tableName'. In the query conditions, deviceId is used as hash key and timestamp is used as range key. These key types are described in Section.2.4.1.1. 'deviceId' is a single integer value identifying the device. 'timestamp' is a string value representing the given date. For example, for the date '2015 Aug 12 14:05', the corresponding string value is '201508121405'.

6.4.2 Cassandra Query Codes

In order to access the Cassandra database, Java API is used. Queries are written in CQL (Cassandra Query Language). Here is a simple Java code for querying a Cassandra table:

```
query = "select * from sensorData where deviceid = ? and year = ?
    and month = ? and day = ? and hour = ? and min = ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device, year,
    month, day, hour, min);
```

This code queries a table named ’sensorData’ with device and time related parameters (year, month, day, hour, minutes). All of these parameters are single integer values representing the id of the device or parameters of the date given.

6.4.3 Apache Spark Query and Processing Codes

In order to connect to the Spark cluster, Java API is used again. Here is a Java code that processes a Cassandra table via a Spark cluster:

```
CassandraJavaRDD<CassandraRow> cRdd = javaFunctions(sparkContext).
    cassandraTable("test", "sensorData");
cRdd = cRdd.where("deviceId=?", device).where("year=?", year);

JavaRDD<Double> rdd = cRdd.map(new MapFunction());

Double result = rdd.reduce(new ReduceFunction());
```

This piece of code queries a table named ’sensorData’ in a keyspace named ’test’. Parameters are id of the device and the year of the data set that will be processed. Both of these parameters are simple integer values.

6.5 Test Results

Tests were performed on both described environments separately. Results are presented in this section and then discussed all together.

6.5.1 NoSQL Database Analysis and Tests

In these tests, two NoSQL databases are tested as described in Section.6.2.a. These tests are applicable to all of the architectures because core part of all of the architectures are NoSQL database, as described in Section.6.1.

6.5.1.1 DynamoDB Analysis and Tests

DynamoDB is extremely easy to deploy. There is no installation required. With just a few clicks, DynamoDB table is ready to be used in a few seconds. Besides ease of installation, there is also no maintenance cost for DynamoDB. As long as user pays for the desired throughput, everything about the database is handled by Amazon. User can modify only throughput of the table after the creation. Even if the user wants, Amazon does not allow user to access internals of DynamoDB. Consequently, in terms of installation and maintenance, DynamoDB is extremely simple for the user.

DynamoDB supports three types of data access methods:

Get operation: This operation allows user to get a single item by specifying the primary key. If the table uses only hash type of primary key, user has to provide only the hash key. However, if the table uses hash and range type of primary key, user has to provide both hash key and range key of the requested item.

Query operation: This operation allows user to request more than one item from the database by specifying a condition. The user can specify the condition on hash and range keys. For hash key, only equality operation can be used. For range key, the user can provide conditional expressions using 'equality', 'greater than', 'less than', 'greater than or equal', 'less than or equal', 'between' and 'begins with' operations. As the result of this operation, items that satisfy the condition returned.

Scan operation: This operation reads all items in a table. By default it returns all items and all of their columns. The user can specify 'ProjectionExpression' to request specific columns and a conditional expression to retrieve items that satisfy the given condition. Scan operation is useful when most of the data in the database will be requested and returned. Amazon's API also supports parallel processing of Scan operation. Scan can be distributed on several threads to iterate concurrently by the client application.

The next criterion is ease of coding a client. Here is a simple Java method that queries a DynamoDB table.

```java
public void query(String tableName, String primaryKey, Integer
    primaryKeyValue, String rangeKey, Long rangeKeyStart, Long
    rangeKeyEnd) throws IOException {
  // initialization
  AWSCredentials awsCredentials = new PropertiesCredentials(this.
      getClass().getResourceAsStream("/AwsCredentials.properties"));
  AmazonDynamoDBClient client = new AmazonDynamoDBClient(
      awsCredentials);
  client.setRegion(Region.getRegion(Regions.US_WEST_2));
  DynamoDB dynamoDB = new DynamoDB(client);
  // end of initialization

  Table table = dynamoDB.getTable(tableName);

  QuerySpec querySpec = new QuerySpec()
     .withHashKey(primaryKey, primaryKeyValue).withConsistentRead(
         false);

  querySpec = querySpec.withRangeKeyCondition(
            new RangeKeyCondition(rangeKey).between(rangeKeyStart,
                rangeKeyEnd));

  ItemCollection<QueryOutcome> items = table.query(querySpec);
  Iterator<Item> iterator = items.iterator();

  while (iterator.hasNext()) {
    Item item = iterator.next();
    // process the item
  }
}
```

As it can be seen in the example code, it is not similar to traditional RDBMS client codes. Still, it is pretty straightforward and simple.

Next, we will test performance of the databases. First parameter is how fast DynamoDB can handle large number of separate insert and read operations. For this part of the test, we will look at the study of Rohan Narde [31]. Details of these tests were given in third paragraph of introduction of this chapter. Insertion times for specified amounts of data are given in Table 6.1. Data record read times are given in Table 6.2. These tests are performed by inserting and reading single data records without caring about values.

Table 6.1: Data insertion performance of DynamoDB

Record Count	1000	10000	50000	100000	500000
Execution Time (seconds)	12.3	23.2	70.3	156.3	634

Table 6.2: Data read performance of DynamoDB

Record Count	1000	10000	50000	100000	500000
Execution Time (seconds)	3.7	14.3	45.3	90	561

The last part of the test is data retrieval speed. Different from the previous test, this test was performed by querying different amounts of data and measuring response times for a single query. Results are given in Table 6.3. The time values are given in milliseconds.

Table 6.3: Execution time of a single query with different amounts of data on DynamoDB

Data Collected in ...	1 year	3 months	1 month	1 week	1 day	5 mins
DynamoDB Execution Time	3878 ms	900 ms	302 ms	74 ms	21 ms	20 ms

Here are codes for querying DynamoDB with different amounts of data:

1-year Data Query:

```java
QuerySpec querySpec = new QuerySpec()
     .withHashKey("deviceId", deviceId)
     .withConsistentRead(false)
     .withRangeKeyCondition(
       new RangeKeyCondition("timestamp")
          .between("201401010000", "201501010000"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

3-months Data Query:

```java
QuerySpec querySpec = new QuerySpec()
     .withHashKey("deviceId", deviceId)
     .withConsistentRead(false)
     .withRangeKeyCondition(
       new RangeKeyCondition("timestamp")
          .between("201401010000", "201504010000"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

1-month Data Query:

```java
QuerySpec querySpec = new QuerySpec()
     .withHashKey("deviceId", deviceId)
     .withConsistentRead(false)
     .withRangeKeyCondition(
       new RangeKeyCondition("timestamp")
          .between("201401010000", "201502010000"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

1-week Data Query:

```
QuerySpec querySpec = new QuerySpec()
    .withHashKey("deviceId", deviceId)
    .withConsistentRead(false)
    .withRangeKeyCondition(
      new RangeKeyCondition("timestamp")
      .between("201401010000", "201501080000"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

1-day Data Query:

```
QuerySpec querySpec = new QuerySpec()
    .withHashKey("deviceId", deviceId)
    .withConsistentRead(false)
    .withRangeKeyCondition(
      new RangeKeyCondition("timestamp")
      .between("201401010000", "201501020000"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

Single Data Query:

```
QuerySpec querySpec = new QuerySpec()
      .withHashKey("deviceId", deviceId)
      .withConsistentRead(false)
      .withRangeKeyCondition(
        new RangeKeyCondition("timestamp")
        .between("201401010000", "201501010005"));

ItemCollection<QueryOutcome> items = table.query(querySpec);
```

According to the results, response time of DynamoDB is directly proportional to amount of the data returned. Since data are returned and transferred in JSON format, as data amount increase, retrieval time also increases linearly.

6.5.1.2 Cassandra Analysis and Tests

Unlike DynamoDB, Cassandra requires substantial amount of effort for installation and maintenance. It must be installed and configured on each node manually. It does not require too much effort for a single node but if there are a lot of nodes, installation will require relatively long time.

Cassandra comes with a tool named ‘nodetool’. This tool allows user to watch the cluster, or a single node. User can send commands to the cluster or a single node like ‘flush data to disk’, ‘enable or disable gossip protocol’, ‘compact a column family’. Thanks to this tool, user does not have to connect each node one by one in order to handle maintenance tasks. Direct access or intervention is required only if a specific node cannot be accessed by nodetool. Consequently, installation, configuration and maintenance of a Cassandra cluster are not as simple as DynamoDB. However, there are utilities that help user to manage the cluster.

For database access, Cassandra provides an SQL like language called CQL. Its syntax is very similar to traditional SQL syntax. Moreover, it supports most of the operations that SQL languages support; even the syntax is the same for most operations. However, a major difference is CQL does not support ‘JOIN’ operation. Other detailed features are described in Section.2.4.2.2.

The next criterion to discuss is ease of coding a client program for Cassandra. Here is a simple Java program that queries a Cassandra table:

```
public void query(int deviceId, int year) {
  Cluster cluster = Cluster.builder().addContactPoint("host-address
      ").build();
  Session session = cluster.connect("test");
  String query = "select * from data where deviceid = ? and year =
      ?";
  PreparedStatement statement = session.prepare(query);
  BoundStatement boundStatement = new BoundStatement(statement).
      bind(deviceId, year);
```

```
    ResultSet resultSet = session.execute(boundStatement);
    Iterator<Row> iterator = resultSet.iterator();

    Row r;
    while (iterator.hasNext()) {
      r = iterator.next();
      double consumption = r.getDouble("consumption");
      // process the data
    }
    cluster.close();
}
```

As it can be seen in example code, interacting with a Cassandra cluster is pretty similar to traditional JDBC clients. Only the connection setup part has a few different parameters and options. In short, a developer with some JDBC experience can easily start coding a client for Cassandra.

Next thing to test is how fast Cassandra can handle large number of separate insert and read operations. Again for this part of the test, we will look at the study of Rohan Narde [31]. Details of these tests were given in third paragraph of introduction of this chapter. Insertion times for specified amounts of data are given in Table 6.4. Read times of data records are given in Table 6.5. These tests are performed by inserting and reading single data records without caring about values.

Table 6.4: Data insertion performance of Cassandra

Record Count	1000	10000	50000	100000	500000
Execution Time (seconds)	3.2	12.7	22.2	55.9	230

Table 6.5: Data read performance of Cassandra

Record Count	1000	10000	50000	100000	500000
Execution Time (seconds)	2	6.2	22.3	51.3	301

The last and the most important part of the test is data retrieval speed. Different from the previous test, this test was performed by querying different amounts of data and measuring response times for a single query. Results are given in Table 6.6. The time values are given in milliseconds.

According to the results, response time of Cassandra is proportional to amount of the data returned, but not very strictly proportional like DynamoDB since it does not transfer data in a format like JSON.

Table 6.6: Execution times of a single query with different amounts of data on Cassandra

Data Collected in ...	1 year	3 months	1 month	1 week	1 day	5 mins
Cassandra Execution Time	1106 ms	320 ms	143 ms	57 ms	16 ms	20 ms

Here are codes for querying Cassandra with different amounts of data.

1-year Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014);
```

3-months Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ? and month >= ? and month < ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014, 2, 5);
```

1-month Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ? and month = ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014, 3);
```

1-week Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ? and month = ? and day >= ? and day < ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014, 4, 5, 12);
```

1-day Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ? and month = ? and day = ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014, 11, 15);
```

Single Data Query:

```
query = "select * from sensorData where deviceid = ? and year
    = ? and month = ? and day = ? and hour = ? and min = ?";
statement = session.prepare(query);
boundStatement = new BoundStatement(statement).bind(device,
    2014, 2, 5, 13, 55);
```

Overall analysis and comparison of the results are presented in Section.6.6.

6.5.2 Concurrent Data Processing Tests

We have performed these tests in order to measure performance of databases in a multi-threaded and multi-process environment. During these tests, the same databases are tested as described in Section.6.2.b. We executed concurrent queries with multiple threads. All of these threads are independent from each other. Query parameters were randomized, in other words none of the threads query same data. Codes for these queries are the same as the ones given in Section.6.5.1. Only difference is in this set of tests, multiple threads execute same code simultaneously. Similar to the previous test, this one is applicable to all of the architectures, since as described in Section.6.1, all of the architectures have data processing component and this component is a must.

6.5.2.1 Data Processing Tests with DynamoDB and Multiple Clients

In these tests, DynamoDB is queried with ten threads to analyze how it responds to simultaneous queries. Moreover, simple calculations are performed on returned data by iterating over it, in order to simulate a data processing environment. Test application is written in Java and deployed to Amazon servers as described in Section.6.3.1, in order to minimize the network delay. In order to do measurements correct, query threads are created first. Then, the counter and threads are started simultaneously. After each thread had exited from processing the query and returned data, the counter is stopped. Results of the tests are given in Table 6.7.

Table 6.7: Execution times of a query with 10 independent threads in parallel and different amounts of data on DynamoDB

Data collected in ... (data size per thread)	1 year	3 months	1 month	1 week	1 day	5 mins
DynamoDB Execution Time (10 threads)	-	31508 ms	745 ms	117 ms	31 ms	3 ms

Since even these results are not good enough to match performance needs of an HVAC EOS, there is no need to perform further tests with higher workloads.

6.5.2.2 Data Processing Tests with Cassandra and Multiple Clients

In these tests, Cassandra is queried with ten threads first. Next, it is queried with twenty threads to examine how it responds to different number of simultaneous queries. In addition to queries, simple calculations are performed on returned data by iterating over it one by one. Test application is written in Java and run on one of the nodes in the cluster that is described in Section.6.3.2. In order to measure time correctly, threads that will perform the query and processing are created first. Then, the counter and threads are started at the same time. After each thread had finished the query and processing and had exited, the counter is stopped.

First, test is performed with ten threads. Results are presented in Table 6.8. Then, the same test is performed with twenty threads. Results are given in Table 6.9.

Table 6.8: Execution times of a query with 10 independent threads in parallel and different amounts of data on Cassandra

Data collected in ... (data size per thread)	1 year	3 months	1 month	1 week	1 day	5 mins
Cassandra Execution Time (10 threads)	1810 ms	513 ms	249 ms	90 ms	34 ms	26 ms

Table 6.9: Execution times of a query with 20 independent threads in parallel and different amounts of data on Cassandra

Data collected in ... (data size per thread)	1 year	3 months	1 month	1 week	1 day	5 mins
Cassandra Execution Time (20 threads)	2030 ms	667 ms	250 ms	100 ms	42 ms	10 ms

6.5.3 Apache Spark Integration

In these tests, selected databases' integration capabilities with distributed data processing frameworks are analyzed. As described before, Apache Spark is used as data processing framework. As database, DynamoDB and Cassandra are used. Again this test is applicable to all of the architectures. Data processing component is present in all of the architectures as described in Section.6.1. When data warehouse is used, it handles data processing while executing query. Therefore, this test is applicable when data warehouse is not used.

6.5.3.1 Data Processing Tests with Spark and DynamoDB

At the moment, Spark does not have internal integration support for DynamoDB. DynamoDB is proprietary product of Amazon, not open source like Spark. There is a few open source attempts to integrate Spark with DynamoDB but they are not stable yet. Without integration, there is a very inefficient method to process DynamoDB data with Spark. First, user has to query database manually. After retrieving the data, it has to be distributed over the Spark cluster using the ‘parallelize’ method of Spark library. Parallelize method distributes the data and creates an RDD (Resilient Distributed Dataset), which is source of all computation in Spark. Then, user can perform any operation on the RDD.

This method is not feasible for two reasons. The first reason is data will be queried by one application and then submitted to the Spark cluster for processing. By this way, user cannot take advantage of possibility of parallel querying. Moreover, as parallel computation load increases, there is a strong possibility that a network bottleneck will arise on the computer that runs the application. The second reason is even if application can retrieve data efficiently under high workloads, the retrieved data have to be distributed over the Spark cluster. This causes another transmission of retrieved data, which will put network in even more stress.

This integration is possible with a larger and Amazon-controlled cluster. For this method we need to deploy an Amazon EMR cluster and connect it to a DynamoDB table. Since we cannot manage internals of this cluster and a much larger cluster is needed, that architecture is out of the scope of this research.

Since there is no internal DynamoDB support in Spark, the problems described above will occur after system works for a while. Therefore, this scenario was not tested in real environment.

6.5.3.2 Data Processing Tests with Spark and Cassandra

Spark has very stable Cassandra integration via external libraries. The most used one is developed by Datastax [12], a commercial Cassandra service provider company. Spark’s Cassandra connector library allows Cassandra tables to be treated as Spark RDDs. By this way, each Spark worker node retrieves only the data that the worker node is going to process. Data do not have to be transmitted to and from a single node multiple times, like user has to do with DynamoDB.

In this test, Spark is connected directly with Cassandra table. Cassandra table is populated with sample data as described in Section.6.3.2. Multiple tests are performed with different amounts of data. Except querying, some processing is also performed on the retrieved data. Here are the time measurements of the tests.

Table 6.10: Execution times of a data analysis on Cassandra using Apache Spark with different amounts of data

Data Collected in ...	1 year	3 months	1 month	1 week	1 day	5 mins
Spark Performance	1393 ms	568 ms	248 ms	126 ms	91 ms	10 ms

According to the results, with small amount of data, Spark brings some overhead compared to tests in Section.6.5.1.2. However, as data amount increases, the performance gap begins to diminish. Data of these tests were not enough to see what happens when data size is much larger. In the next test, this condition will be examined.

Here is an example code for processing a Cassandra table with Spark:

```
CassandraJavaRDD<CassandraRow> cRdd = javaFunctions(sparkContext)
                        .cassandraTable("test", "sensorData");
cRdd = cRdd.where("deviceId=?", device).where("year=?", 2014);

JavaRDD<Double> rdd = cRdd.map(new MapFunction());

Double result = rdd.reduce(new ReduceFunction());
```

6.5.4 Data Processing Tests with Large Datasets

In previous test, results showed that with small amount of data, distributed processing brings some overhead. During these tests, much larger data sets will be processed in order to examine effects of distributed data processing frameworks. We have used specific tools that can work together and integrated. Therefore, this test is for any architecture described in Section.6.1 if Apache Cassandra and Spark are used. Without tools that can work together, data would be queried and distributed manually and these operations bring a lot of data movement and overhead to the system.

This test will be conducted in three steps. Data always reside on Cassandra. First, data will be retrieved by a client application and processing will be done in the application manually. Then, data processing will be performed with Apache Spark with two different configurations. In the first configuration, Spark cluster consists of four computers and processing is performed in four nodes concurrently. In second configuration, there is only one Spark node but it processes data with four threads. Data are processed by four nodes concurrently like in previous configuration, but this time they are on the same computer.

In previous tests, the largest data set that was processed was a single device's data of one year. During this test, all of the data present in the table are used. In other words, there are data of fifty devices. For each device there are data for two years. This makes a hundred times more data than previous tests.

Table 6.11: Execution times of the processing of the whole database on Cassandra with and without Apache Spark

Test	Duration
Cassandra execution time	94052 ms
Spark with 4 nodes execution time	75932 ms
Spark with 1 node 4 threads execution time	71878 ms

According to the in results, distributed processing frameworks bring significant improvement to applications that process large amount data. However, they are not beneficial for all systems; processed data size has to be over some threshold.

6.6 Comments on Results

In Table 6.12, summary of all tests are displayed. Comments on this table are given in the following sections. In Table 6.13, proposed architectures and tests that are related or applicable to them are given. In Table 6.14, proposed architectures and their properties are given. Some of the items are marked with both positive and negative. This means property depends on the tool choice. For example if DynamoDB is used as NoSQL database, Apache Spark integration test is not applicable. However, if Cassandra is used, then that test makes sense.

Table 6.12: Summary of results of all tests (Execution times with different data amounts)

Data Collected in ...		1 year	3 months	1 month	1 week	1 day	5 mins	2 years with 50 sensors (Whole Database)
Single Client	DynamoDB	3878 ms	900 ms	302 ms	74 ms	21 ms	20 ms	-
	Cassandra	1106 ms	320 ms	143 ms	57 ms	16 ms	20 ms	-
Multi Client	DynamoDB (10 Threads)	-	31508 ms	745 ms	117 ms	31 ms	3 ms	-
	Cassandra (10 Threads)	1810 ms	513 ms	249 ms	90 ms	34 ms	26 ms	-
	Cassandra (20 Threads)	2030 ms	667 ms	250 ms	100 ms	42 ms	10 ms	-
Spark Integration	DynamoDB	-	-	-	-	-	-	-
	Cassandra	1393 ms	568 ms	248 ms	126 ms	91 ms	10 ms	-
Large Dataset	Cassandra	-	-	-	-	-	-	94052 ms
	Spark with 4 nodes	-	-	-	-	-	-	75932 ms
	Spark with 1 node and 4 threads	-	-	-	-	-	-	71878 ms

Table 6.13: Architecture-Test Applicability

Architectures	Tests Applicability			
	NoSQL Analysis & Tests	Concurrent Data Processing	Distributed Processors Integration	Large Dataset Processing
NoSQL DB & Data Processor	+	+	+ with Spark only	+ with Spark only
NoSQL & In-Memory with Data Processor	+	+	+ with Spark only	+ with Spark only
Data Warehouse	+	-	-	-
Data Stream Handler	+	+	+ with Spark only	+ with Spark only
Distributed Data Stream Processor	+	+	+ with Spark only	+ with Spark only

* (+) means the test in the column header is meaningful for or applicable to the architecture. (-) means the opposite.

Table 6.14: Architecture Properties

Architectures	Properties			
	Distributed Data Processing	Query Processing (Data Warehouse)	Data Stream Processing	In-Memory Storage
NoSQL DB & Data Processor	+ with Spark only	-	-	-
NoSQL & In-Memory with Data Processor	+ with Spark only	-	-	+
Data Warehouse	+ with Spark only	+	-	+ with In-Memory DB only
Data Stream Handler	+ with Spark only	+	-	+ with In-Memory DB only
Distributed Data Stream Processor	+	+	+	+ with In-Memory DB only

* (+) means the architecture provides the property of that column. (-) means the opposite.

6.6.1 Comments on NoSQL Database Analysis and Tests

According to results of first set of tests, both databases have its pros and cons. Three criterions were examined during these tests: ease of installation and maintenance, data access methods and ease of provided API and data retrieval speed.

As stated before, Cassandra requires some effort for installation and maintenance of nodes in the cluster while DynamoDB requires almost no effort. DynamoDB handles everything behind the curtains after user specifies desired throughput. Therefore, DynamoDB is certain winner in terms of installation and maintenance effort.

In terms of provided operations, they provide similar operations. User can query both databases by specifying conditions based on primary or secondary keys. DynamoDB provides a different method to access data, scan operation. Scan operation is not something to be used frequently since it searches a whole table record by record. Therefore, none of the databases is superior to the other one in terms of provided operations.

In terms of data access method, Cassandra provides a query language called 'Cassandra Query Language (CQL)' while DynamoDB does not have something like that. User has to write code in Java or some other language that Amazon supports. Moreover, Cassandra has a tool called 'cqlsh' which is a console application like MySQL console client. Using cqlsh, users can query and modify tables and insert into tables by writing simple CQL scripts. However, for DynamoDB, user has to write code or use Amazon's DynamoDB web interface. In terms of data access methods, Cassandra is slightly easier to learn and develop since its tools and language is similar to traditional ones.

For programming purposes, both databases provide simple API's. User can start development with very little exercise. However, since Cassandra has a separate language for database operations, code of a Cassandra client is very similar to code of a traditional RDBMS, which makes learning curve pretty gradual. In addition, since queries are written in constant strings, it is easier to read and understand the code of a Cassandra client. While both databases provide simple API's, DynamoDB API has a little steeper learning curve and code is a little harder to read than Cassandra API.

As described in Section.6.5.1, according to Rohan Narde's study, Cassandra can execute two times more read operations than DynamoDB in same time interval. In terms of write performance, Cassandra executes about three times more operations in average. In his study, he did not analyze data format and do any configurations for a fixed data format. In our tests, we have used time-series data and defined its format. In addition, we have configured the database and its indexes accordingly. Consequently, with a few data records, there is not much difference between two databases but as data size increases, Cassandra is about three times faster with our configuration. As we can see in Table 6.12, performance gap further increases as data size grows and probably will be much more than only three times if we further increase data size. Rohan Narde claims Cassandra is faster and we claim it can be even faster with correct configuration and our tests validate this. About data retrieval speed, Cassandra is the clear winner according to tests. Still both of them have very good performance for working with large amounts of data. In these tests, environments are configured for best performance. We were able to deploy Cassandra to a local cluster. So we were able to tune the system for higher Cassandra performance and even without any tuning, communication delay was very low. For DynamoDB, we do not know how things are handled in Amazon's systems.

Except raw time measurements, response time of DynamoDB is directly proportional to amount of the data returned while response time of Cassandra is not very strictly proportional like DynamoDB. The most probable reason of this is DynamoDB transfers data in JSON format. This has two main unwanted side effects. First one is there are lots of redundant data transferred with each item, for example column names, especially when transferred item count is large. Second side effect is each item contains table metadata even if the table structure is fixed. Since Cassandra knows table metadata even if some columns are not used, it can utilize transfer of this information.

All of the results are enough to fulfill performance requirements given in Chapter 4. Results of this test can be used while designing any system that includes a NoSQL database. All of our proposed architectures have NoSQL database as the storage component (Section.6.1), therefore whatever architecture we choose, we are going to keep these results in mind while deciding tools.

6.6.2 Comments on Concurrent Data Processing Tests

DynamoDB and Cassandra are tested in an environment similar to a real energy tracking and optimization system. First, DynamoDB is queried with ten threads concurrently that perform simple processing on data. Then, Cassandra is queried with ten threads and then twenty threads. DynamoDB was not tested with twenty threads since performance of ten threads was enough to compare with Cassandra. These threads are independent from each other and query same amount of data.

For Cassandra, there is not much performance difference between ten and twenty threads. In both tests, measured duration is about twice the time of single-threaded tests for each tested data amount.

For DynamoDB, results are not as linear as Cassandra. When data amount is small, response time is almost the same as single-threaded DynamoDB tests. However, as data amount increases, response time begins to increase very rapidly. When we had executed queries exactly at the same time, DynamoDB had a hard time handling requests efficiently and results are very high as we see in tables. Then, we introduced a few milliseconds between each query execution and this time results were much better but still worse than Cassandra. This anomaly is probably because of internal request handling and request rate throttling but it is impossible for us to track the problem.

Cassandra is faster in all tests, which may be because of running on a local cluster. If we consider the increase rate in response times according to test results, when database interactions include very small amount of data, DynamoDB's performance is more stable. Its performance in multi-threaded environment is very close to single-threaded tests. When data sets are small, both databases meet the performance requirements given in Chapter 4, but DynamoDB is slightly faster. When data amount is larger, Cassandra is more stable and meets the performance requirements while DynamoDB begins to suffer from performance and stability problems.

Results of this test can be used to design any system with NoSQL database. Our architectures also use NoSQL database as storage and concurrent processing is a must (Section.6.1). Thus, these results will guide us while designing our final architecture.

6.6.3 Comments on Apache Spark Integration Tests

Apache Spark integration of the tested databases was examined and a few calculation tests are performed. First, Spark was integrated with a Cassandra table and tests were performed on data sets with different sizes. When data amount is small, adding Spark to data processing stack brings a lot of overhead. As data amount increase, the performance gap caused by Spark diminishes and results expressed in Section.6.5.3.2 show what happens when data size further increases.

DynamoDB does not have direct integration with Spark. It is possible to query data from DynamoDB manually and distribute it over the Spark cluster for processing. However, even query speed of DynamoDB is slower than Spark processing on Cassandra. Moreover, this test would not be feasible to be used in a real working system because of the reasons described in Section.6.5.3.1.

Results of this test will guide us while deciding which data processing method to use. As described in Section.6.1, we can either use simple data processing application or distributed BigData processing frameworks in all of the architectures. If data warehouse is used in the system, it can handle distributed processing of data while querying. Thus, there is no need to use Apache Spark with data warehouse. This test is applicable only when no data warehouse is used in the system. According to the test results, if underlying NoSQL database is Cassandra, Apache Spark can be integrated easily. We are going to discuss impact of Spark on data processing on next section.

6.6.4 Comments on Processing of Very Large Datasets

We have seen that Apache Spark brings overhead when data amount is small and performance gap closes as data amount increase. In order to see Spark's effect in larger data sets, same tests were executed on the whole data residing on the database.

First, the whole data were processed without Spark. Then, same process was performed with four Spark nodes. After that it was performed with one Spark node running on four threads. According to the results, as data size gets larger, benefits of the Spark get much more than its overhead. It is clear that parallel processing power of Spark saves significant amount of processing time for large data sets.

During the tests, Spark with one node and four threads performed slightly better than Spark with four nodes. This is because of the network overhead introduced with multiple nodes. Moreover, total data size was enough to be kept in memory; therefore, all four threads can perform processing completely on memory of a single node. If memory of a node was not enough to hold the whole data, Spark with four nodes would perform better since more data can be processed in memory simultaneously.

Results of this test reveal impact of Apache Spark integration on data processing. According to the Section.6.1, data processor is a critical component for all of the architectures. We are going to consider these results while choosing data processing component for recommended architecture.

6.7 Recommended System Architecture

After analyzing the test results, the recommended final system architecture is shown in Figure 6.1.

Although the deployment and maintenance costs are higher, a local Cassandra cluster was chosen as database for its speed and limitless data size. In-memory database is not necessary for the final system since data size is very large and operations that are related to user interactions are performed by in-memory data processing framework, Apache Spark. In addition to Apache Spark, Apache Hive is used for operations that are performed behind the curtains and process very large amounts of data. It is mostly for data sets that do not fit in total memory of the Spark cluster. Data processor node is responsible from map-reduce task submissions and communication between the HVAC EOS and BMS.

Stream processing is used in the system because it takes a great burden from the NoSQL database and data processor. Data preprocessing, watching building demand and alarms & notifications operations are performed in stream processing pipeline. In this pipeline, both Amazon Kinesis and Apache Storm were used. Amazon Kinesis is

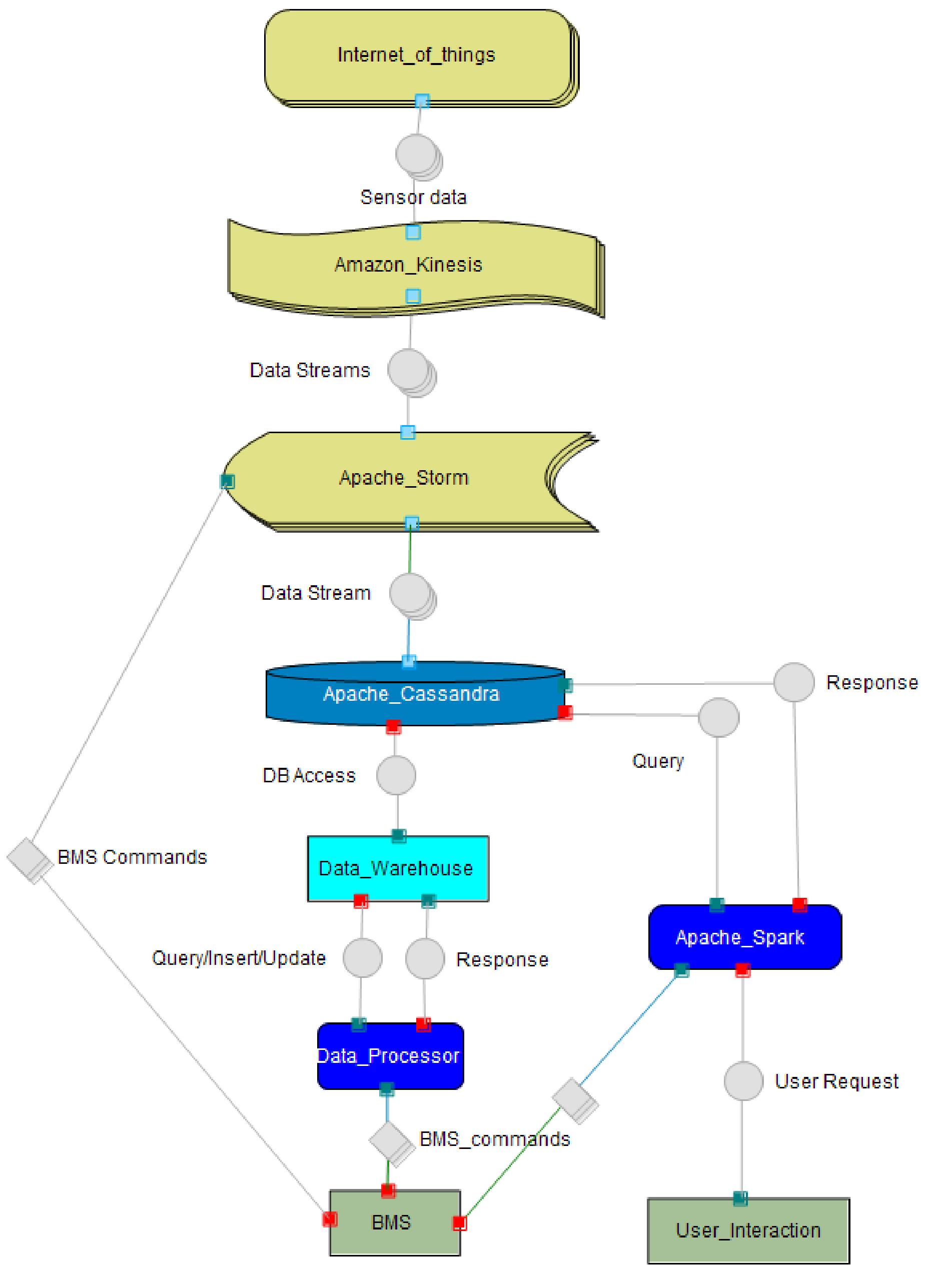


Figure 6.1: Recommended System Architecture

the first arrival point for the data. Data are stored there until a Storm worker retrieves it. Kinesis is mainly used for storing large amount of data streams. If Kinesis was not used, if at some point Storm cluster gets slower, data generated by the sensor may be lost because a Storm worker could not retrieve them and data were overridden by the sensor. Therefore, any data generated by the sensors are stored directly in Kinesis without any processing. Then, Storm cluster performs preprocessing of the data, takes immediate actions depending on the data if necessary and writes it into Cassandra.

6.7.1 Operations Met by the Components of the Recommended System

In this section, we will present which modules of the final system is responsible from the operations described in Section.4.3.

Static Building Energy Performance Simulation: As described in Section.4.3.1, this operation is performed once for each building. Inputs of this operation are static values therefore there is no BigData here. This operation is performed by the data processor node in the final architecture.

Data Preprocessing: This operation is the first one in the pipeline that processes the BigData. As described in Section.4.3.2.1, steps of this operation are independent from past or future data. Therefore, each data packet can be processed individually. For this purpose, data stream processing pipeline (Amazon Kinesis and Apache Storm) is used to handle this operation. Each data unit is processed one by one as it propagates through the pipeline before it is written to the database. In terms of performance, this operation depends on only propagation delay of data packets through the network since there is no complex calculations or large data-sets at this point.

Watching Building Demand: This operation processes only recent (15 minutes) demand data of a sensor or a set of sensors as described in Section.4.3.2.2. Therefore, there is no need to query the database frequently and impose a large burden to the database. These data can be processed in the data stream processing pipeline by keeping only small amount of summary data about recent demand data. Current data and summary of recent data are combined and analyzed together. Similar to previous operation, in terms of performance, this operation depends on only propagation delay of data packets through the network since there is no complex calculations or large data-sets at this point of the system.

Forecasting: This operation requires large amount of past data as described in Section.4.3.2.4, so it has to be performed by one of the map-reduce clusters. When total data amount is not very large, Apache Spark is used for speed. In our reference

EOS, the largest data set is one-year data of a sensor, which can fit easily in the memory of a cluster of a few nodes. However, as system grows, larger analyzes may be needed with much larger data-sets or there may be too many simultaneous forecasting operations for different buildings. In situations like this, processing will be performed on Apache Hadoop via data warehouse (Apache Hive) since it stores data on disk during operations.

BMS Optimization: As described in Section.4.3.2.5, calculations in this operation are similar to forecasting. Used data amount, input data types and processing algorithm details differ slightly. Therefore, this operation is going to be performed at the same point and same method with forecasting operation.

Alarms and Notifications: This operation has to be performed on each data packet since its purpose is to notify the user immediately for predefined alarms as described in Section.4.3.2.3. Operation style is similar to data preprocessing operation. Therefore, this operation is also performed on data stream processing pipeline similar to data preprocessing operation. For each data unit, alarm checks are going to be performed and notifications are triggered when needed. In terms of performance, this operation depends on only propagation delay of data packets through the network since there is no complex calculations or large data-sets at this point of system.

Table 6.15: System Components and Matched Operations

System Operation	Components			
	Data Stream Processing Pipeline	NoSQL Database	Data Warehouse	Distributed Data Processing Framework
Data Preprocessing	X			
Watching Building Demand	X			
Forecasting			X	X
BMS Optimization				X
Alarms & Notifications	X			
Data Storage		X		

CHAPTER 7

CONCLUSION

Systems that produce BigData also produces several challenges because of the characteristics of the data, like its volume, velocity or variety. Traditional data storage and processing systems are mostly scalable vertically. However, this is very expensive after some point and hard to add more power because of hardware limits. This thesis analyzes a BigData analytics system and proposes horizontally scalable solutions to problems of BigData.

In this thesis, we have analyzed needs of a HVAC EOS, which is one of the important areas of energy domain. Detailed description of its requirements and operations are presented formally. Then, technologies that can be used to fulfill the requirements were examined. After defining requirements, different software architectures were recommended in order to fulfill various needs of the system. Then, some of the recommended tools were tested in different environments, one is a local cluster and the other one is a cloud platform. Then, test results are shared, analyzed and comments on the results are expressed. Finally, an optimal system architecture was proposed in consideration of the test results and tool characteristics.

According to results, we arrived several conclusions in addition to the recommended system architecture:

- A local cluster performs better than cloud environment when communication speed is critical.
- Scaling a local cluster is harder and more expensive than cloud environment since physical hardware need to be installed and maintained manually. Scal-

ing a cloud cluster is much easier since everything needed for expanding it is provided by the cloud service provider.

- Maintaining a cloud-based cluster is much easier than maintaining a local one. User only specifies the system properties (hardware and software needs) and they are provided by the cloud service provider. However, in a local cluster, user has to install and manage every hardware and software manually.
- Column-based NoSQL databases are more suitable for data with large item size.
- Integration of the databases with a distributed data processing framework increases performance greatly when processed data amount is large.
- In order to respond quickly to rapid changes, data needs to be processed on the fly, preferably with data stream processing techniques.

Results of the study enables us to identify needs of a system and choose a suitable architecture. Moreover, the study gives a formal definition of a HVAC EOS, so that operations of an IoT system are exemplified. In addition, pros and cons of architectures are explained and related tools are tested and analyzed. With detailed definition of BigData and NoSQL technologies, the thesis gives insight to BigData concepts.

There are a few points that can be improved on this thesis. First one is testing of the tools and architectures in identical environments. During the tests, the environments are configured for the best performance. One of the databases (DynamoDB) was tested in cloud environment completely since it can be deployed only on Amazon's cloud. We might have received different results if tests were performed in identical environments. Second improvement would be constructing a testing environment very similar to a real system and perform complete architecture tests on this environment. In this thesis, a few key tools of the architectures are tested and analyzed. The proposed architectures should be installed and configured as described in the thesis and then tests should be performed. As the last and the most important future work, data anomaly detection and machine learning algorithms used in a HVAC EOS can be studied. This thesis does not give any details of the algorithms used in the system. Besides the BigData management, defining and optimizing these algorithms are other important works to be done.

REFERENCES

[1] Amazon Kinesis. http://aws.amazon.com/kinesis/. Accessed: 2015-06-30.

[2] Apache Cassandra. http://hbase.apache.org/. Accessed: 2015-06-30.

[3] Apache CouchDB. http://couchdb.apache.org/. Accessed: 2015-06-30.

[4] Apache Hadoop. https://hadoop.apache.org/. Accessed: 2015-06-30.

[5] Apache HBase. http://hbase.apache.org/. Accessed: 2015-06-30.

[6] Apache Hive. https://hive.apache.org/. Accessed: 2015-06-30.

[7] Apache Spark. https://spark.apache.org/. Accessed: 2015-06-30.

[8] Apache Storm. https://storm.apache.org/. Accessed: 2015-06-30.

[9] The cap theorem. https://foundationdb.com/key-value-store/white-papers/the-cap-theorem. Accessed: 2015-06-30.

[10] Cometcloud. http://nsfcac.rutgers.edu/CometCloud/. Accessed: 2015-06-30.

[11] Data stream processing explained. http://www.sqlstream.com/stream-processing/. Accessed: 2015-08-10.

[12] Datastax. http://www.datastax.com/. Accessed: 2015-06-28.

[13] Dynamodb. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. Accessed: 2015-06-30.

[14] Energyplus. http://apps1.eere.energy.gov/buildings/energyplus/. Accessed: 2015-06-30.

[15] Mongodb. https://www.mongodb.org/. Accessed: 2015-06-30.

[16] National cable and telecommunications association. https://www.ncta.com/platform/industry-news/infographic-the-growth-of-the-internet-of-things/. Accessed: 2015-08-03.

[17] Redis. `http://redis.io/`. Accessed: 2015-06-30.

[18] Riak. `http://basho.com/`. Accessed: 2015-06-30.

[19] Using amazon kinesis with apache storm. `https://aws.amazon.com/blogs/aws/marry-kinesis-and-storm-using-the-new-kinesis-storm-spout/`. Accessed: 2015-08-10.

[20] Kenneth P Birman, Lakshmi Ganesh, and Robbert Van Renesse. White paper running smart grid control software on cloud computing architectures. *Computational Needs for the Next Generation Electric Grid*, 2011.

[21] Cisco Visual Networking Index Cisco. Global mobile data traffic forecast update, 2013–2018. *white paper*, 2014.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Zhiming Ding, Jiajie Xu, and Qi Yang. Seaclouddm: a database cluster framework for managing and querying massive heterogeneous sensor sampling data. *The Journal of Supercomputing*, 66(3):1260–1284, 2013.

[24] Xi Fang, Satyajayant Misra, Guoliang Xue, and Dejun Yang. Managing smart grid information in the cloud: opportunities, model, and applications. *Network, IEEE*, 26(4):32–38, 2012.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[26] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[27] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[28] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[29] Hongseok Kim, Young-Jin Kim, Kai Yang, and Marina Thottan. Cloud-based demand response for smart grid: Architecture and distributed algorithms. In *Smart Grid Communications (SmartGridComm), 2011 IEEE International Conference on*, pages 398–403. IEEE, 2011.

[30] Tingli Li, Yang Liu, Ye Tian, Shuo Shen, and Wei Mao. A storage solution for massive iot data based on nosql. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 50–57. IEEE, 2012.

[31] Rohan Narde. *A Comparison of NoSQL systems*. PhD thesis, Rochester Institute of Technology, 2013.

[32] Ioan Petri, Omer Rana, Yacine Rezgui, Haijiang Li, Tom Beach, Mengsong Zou, Javier Diaz-Montes, and Manish Parashar. Cloud supported building data analytics. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 641–650. IEEE, 2014.

[33] Sebnem Rusitschka, Kolja Eger, and Christoph Gerdes. Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 483–488. IEEE, 2010.

[34] John Shonder. Facts sheets on hvac measures, 2010.

[35] Yogesh Simmhan, Saima Aman, Alok Kumbhare, Rongyang Liu, Sam Stevens, Qunzhi Zhou, and Viktor Prasanna. Cloud-based software platform for big data analytics in smart grids. *Computing in Science & Engineering*, 15(4):38–47, 2013.

[36] Yogesh Simmhan, Alok Gautam Kumbhare, Baohua Cao, and Viktor Prasanna. An analysis of security and privacy issues in smart grid software architectures on clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 582–589. IEEE, 2011.

[37] Christopher Surdak. *Data Crush: How the Information Tidal Wave is Driving New Business Opportunities*. AMACOM Div American Mgmt Assn, 2014.

[38] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.

[39] Bogdan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[40] Sapna Tyagi, Ashraf Darwish, and Mohammad Yahiya Khan. Managing computing infrastructure for iot data. *Advances in Internet of Things*, 2014, 2014.

[41] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101, 2012.

[42] Melike Yigit, V Cagri Gungor, and Selcuk Baktir. Cloud computing for smart grid applications. *Computer Networks*, 70:312–329, 2014.