An Experimental Performance Comparison of NoSQL and RDBMS Data Storage Systems in the ERP System Odoo

Author: MHD Fawaz Enaya

November 28, 2016

Advisors:
Dr. Dipl.-Wirt.-Inf. Frederik Kramer
InitOS

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Andreas Meister
Otto-von-Guericke Universität Magdeburg
Enaya, MHD Fawaz:
An Experimental Performance Comparison of NoSQL and RDBMS Data Storage Systems in the ERP System Odoo
Abstract

Odoo is an open source, Enterprise Resource Planning (ERP) system. During the lifetime of a running Odoo system, its performance is decreasing due to the growing amount of data. Mail messages and attachments modules are special modules in Odoo. These two modules are used by almost all other Odoo modules. Accordingly, their stored data is getting larger faster than other modules. This problem reduces Odoo performance. In this thesis, we try to improve the latency of Odoo system which is caused by mail messages and attachments modules.

In the last decade, new data storage systems have been developed, so-called Not Only SQL or Non SQL (NoSQL) systems. NoSQL systems have been developed to solve Relational Database Management Systems (RDBMSs) scalability problems and to be more efficient systems for managing the big data.

To solve Odoo latency problem, we suggest to store the data of mail messages and attachments modules in NoSQL data storage, and modify Odoo to communicate with this new data storage system.

After we customized Odoo, we made an experiment to compare the performance between regular Odoo system and our modified Odoo system. The results of this experiment indicate that, for small and medium size companies, Odoo performs better without using NoSQL system. If the amount of data of the mail messages and the attachments module is huge, our modified version of Odoo is faster.
Acknowledgements

This thesis is the end of my academic journey to get my master degree from Otto-von-Guericke University, Magdeburg. I would like to thank all academic and office stuff in the university, and all the people who were with me the last two years.

First and foremost, I would like to thank my adviser Prof. Gunter Saake, for giving me the opportunity to make this work under his supervision. I would like also to thank Prof. Gunter Saake, for his guidance during the work and for his valuable lectures that I learned a lot from and encouraged me to work more with database topics.

I would like to express my thanks and gratitude for my adviser Dr. Frederik Kramer, for his guidance to select the research topic as well as his contribution of knowledge during the work. Also, I would like to thank all InitOS GmbH employees especially Dr. Thomas Rehn for their support during my employment time at InitOS GmbH.

I would like to give a special thank to M.Sc. Andreas Meister for his comprehensive support and his helpful remarks and suggestions during the writing of this thesis.

There is no words to express my gratitude to my father and my mother, who always supported and encouraged me. Furthermore, I must express my sincere thanks to my wife and my son for their patient and endless support. Finally, I would like to thank my family and all my friends who continuously assist me and always with me.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Code Listings</td>
<td>xv</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>xvii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Enterprise Resource Planning Systems</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 The Types of ERP Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Available ERP Systems</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Odoo</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Mail Messages Module:</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Attachment Module:</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3 Usage of Mail Message and Attachment Modules:</td>
<td>16</td>
</tr>
<tr>
<td>2.3 NoSQL Database Storage</td>
<td>18</td>
</tr>
<tr>
<td>2.3.1 BASE Characteristics for NoSQL</td>
<td>18</td>
</tr>
<tr>
<td>2.3.2 SQL v.s. NoSQL</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3 NoSQL Data Model</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Hadoop Ecosystem</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1 Hadoop</td>
<td>27</td>
</tr>
<tr>
<td>2.4.2 Zookeeper</td>
<td>29</td>
</tr>
<tr>
<td>2.4.3 HBase</td>
<td>30</td>
</tr>
<tr>
<td>2.4.4 Phoenix</td>
<td>31</td>
</tr>
<tr>
<td>3 Design and Implementation</td>
<td>33</td>
</tr>
<tr>
<td>3.1 Requirement Analysis</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Implemented Hadoop Stack</td>
<td>34</td>
</tr>
<tr>
<td>3.3 HBase ORM Odoo Module</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1 Data Model</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2 Class Diagram</td>
<td>40</td>
</tr>
<tr>
<td>3.3.3 Read, Create, Update and Delete</td>
<td>41</td>
</tr>
</tbody>
</table>
3.3.4 Search ................................................................. 44
3.4 System Architecture ................................................. 45

4 Evaluation ................................................................. 47
  4.1 Hadoop Cluster Implementation ................................. 47
  4.2 HBase Client API Evaluation .................................... 49
  4.3 Data Characteristics ................................................ 49
  4.4 Unit Test Evaluation ............................................... 50
  4.5 Front-end Evaluation .............................................. 52
  4.6 Database Systems Evaluation ..................................... 57

5 Conclusion ............................................................... 61

6 Future Work ............................................................ 63

A Appendix ................................................................. 65

Bibliography ............................................................... 69
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Hadoop Cluster</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Hadoop Data Distribution</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Unit Test Successful Result</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>User’s New Messages</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>User’s Sent Messages</td>
<td>53</td>
</tr>
<tr>
<td>4.6</td>
<td>Processing Time for View Messages Page</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>Processing Time for Filtering the Messages on Title</td>
<td>55</td>
</tr>
<tr>
<td>4.8</td>
<td>Modified Odoo Front-End Requests</td>
<td>56</td>
</tr>
<tr>
<td>4.9</td>
<td>Regular Odoo Front-End Requests</td>
<td>56</td>
</tr>
<tr>
<td>4.10</td>
<td>Filter Messages on Body Content</td>
<td>57</td>
</tr>
<tr>
<td>4.11</td>
<td>Phoenix Execution Plan</td>
<td>58</td>
</tr>
<tr>
<td>4.12</td>
<td>Filter Messages on Body Content</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Mail Message Data Example ............................................. 14
3.1 Mail Message De-normalization ......................................... 38
4.1 Cluster Nodes Specification .............................................. 48
4.2 PostgreSQL Server Specification ....................................... 48
4.3 Sample Data Record Counts .............................................. 49
### List of Code Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Code Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Integrating Mail Message with Product</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Document Oriented Databases</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Odoo Model Create Function</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Odoo Model Read Function</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Search Attachment Unit Test</td>
<td>51</td>
</tr>
<tr>
<td>A.1</td>
<td>Hadoop core-site.xml</td>
<td>65</td>
</tr>
<tr>
<td>A.2</td>
<td>Hadoop hdfs-site.xml</td>
<td>66</td>
</tr>
<tr>
<td>A.3</td>
<td>HBase Configuration hbase-site.xml</td>
<td>67</td>
</tr>
</tbody>
</table>
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation, Durability</td>
</tr>
<tr>
<td>AJAX</td>
<td>Asynchronous JavaScript and XML</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update, Delete</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>ERD</td>
<td>Entity Relationship Diagram</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>GmbH</td>
<td>Gesellschaft mit beschränkter Haftung</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HBase</td>
<td>Hadoop Database</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HR</td>
<td>Human Resource</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>NoSQL</td>
<td>Not Only SQL or Non SQL</td>
</tr>
<tr>
<td>OLAP</td>
<td>Online Analytical Processing</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing</td>
</tr>
<tr>
<td>ORDBMS</td>
<td>Object-Relation Database Management System</td>
</tr>
<tr>
<td>ORM</td>
<td>Object Relational Mapper</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>OSS</td>
<td>Open-source Software</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SAP</td>
<td>Systems, Applications &amp; Products</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>YAML</td>
<td>Yet Another Markup Language</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator</td>
</tr>
</tbody>
</table>
1. Introduction

Odoo is an open source ERP system which contains variety of applications, such as: Accounting, inventory management, customer relationship management and many other applications. These applications work consistently with each other’s to manage companies of all sizes. The application in Odoo is made up of one or several Odoo modules. Odoo is built to work tightly with PostgreSQL\(^1\) as Object-Relation Database Management System (ORDBMS), with time and increasing amount of data stored in PostgreSQL database the performance of the system will be reduced, which leads to bad customer experience.

InitOS Gesellschaft mit beschränkter Haftung (GmbH)\(^2\) noticed a slow performance problem in an Odoo system, which was developed for one of its customers. Analyzing the problem revealed that, the problem is caused by a large size of tables related to Attachments and Mail messages modules as they grow rapidly during the lifetime of the system.

To solve the problem, we can:

- Archive the old data to a data warehouse storage.
- Use different database management system that supports managing and processing huge amount of data.

The first solution solves the inefficient problem by reducing the amount of data stored in the database. However, this solution creates other problems. It will prevent Odoo from accessing the old archived data, and generate overwork when we need to apply data processing tasks on the whole data set. Then, we need to perform two separated

---

\(^1\)https://www.postgresql.org/about/

\(^2\)http://www.initos.com/
1. Introduction

tasks: One for the archived data and the other for the live data. Besides, most likely the system will suffer from overhead load while the data processing tasks are running.

In this thesis, we choose to work with the second solution by extending Odoo to support communicating with data storage system other than PostgreSQL.

In the last years, it becomes more common for companies to process Petabytes\(^3\) or Exabytes\(^4\) of data. To cope with this new requirement, which is beyond the capabilities of normal RDBMS, the computer scientists developed many new database storage systems [MH13]. These new systems have been grouped under the name NoSQL.

NoSQL data storage system may overcome the above described problems.

The aim of this thesis is:

- To solve the bad performance of Odoo system, which is caused by mail messages and attachments modules, via using NoSQL data storage to store mail messages and attachments data.

This goal can be achieved by the following steps:

- We need to select a NoSQL data storage system that satisfies our requirements.
- We have to develop an Odoo module that enables Odoo Object Relational Mapper (ORM) to communicate with NoSQL database system.
- We should compare the performance between the new system and the regular Odoo system to evaluate our system.

We choose to use Hadoop ecosystem as the new data storage system for Odoo. Hadoop ecosystem is a set of distributed applications and systems which are built to manage, store, search, query and process huge amount of variety data. Each application in the Hadoop ecosystem focuses on specific data problem to solve. We need to select the applications that help us achieving our goal.

The remainder of this thesis is structured as following:

- **Chapter 2 Background:** In this chapter, we explain the problem that we try to solve in more details. Then, we describe Odoo modules which we extend to support NoSQL Database Management System (DBMS). After that, we illustrate the Hadoop ecosystem’s applications that we use.

---

\(^3\) One Petabyte equals \(10^{15}\) bytes

\(^4\) One Exabyte equals \(10^{18}\) bytes
• **Chapter 3 Design and Implementation:** At first, we describe the system requirements. Then, we illustrate the benefits of the Hadoop components that we used. Finally, we explain how we implemented the new Odoo module, and we show the changes that we applied on current mail messages and attachments modules.

• **Chapter 4 Evaluation:** In this chapter, we show the evaluation environment and present our experiment results for front-end and database performance analysis.

• **Chapter 5 Conclusion:** In this chapter, we summarize our experiment steps and its results.

• **Chapter 6 Future Work:** Finally, we suggest how we can enhance Odoo performance more and what Odoo can benefit from our new module.
2. Background

In this chapter, we provide the theoretical background about the concepts we use within this thesis. In Section 2.1, we define the ERP systems, their components and types. Then, in Section 2.2, we illustrate Odoo as an open source ERP system and explain Odoo modules\(^1\) that we modified in our developed systems. After that, we demonstrate the NoSQL database systems and their data models. Then, we compare it to relation Structured Query Language (SQL) database systems in Section 2.3. Since we use Hadoop ecosystem as our NoSQL data storage system, in Section 2.4, we describe this system and explain the components which we use in our implementation.

2.1 Enterprise Resource Planning Systems

Finding the right information in the right time to take a good decision is an essential feature for any successful and competitive organization, and such decision could not be made unless the organization’s management has an overall knowledge about their organization’s departments. This requirement is tough to be realized if each functional department of the organization has its own information system. Instead, if the organization can manage all these departments using one system that would be more effective and will open new opportunities for the organization. The ERP systems provide such integrated management for all businesses.

The ERP systems can be defined as “commercial software packages promise the seamless integration of all the information flowing through a company—financial and accounting information, human resource information, supply chain information, customer information.” (Davenport, 1998) [Dav98].

ERP systems help organizations to manage all aspects of their business in a central system, for example:

\(^1\)Module is the main building block in Odoo to develop any functionality
In Figure 2.1, we show some of the modules that may build up an ERP system. Each organization selects the modules that should form its ERP system based on the organization’s business processes.

2.1.1 The Types of ERP Systems

ERP systems can be categorized based on different considerations, such as:

- The system architecture (2-tier, 3-tier, n-tier) [LH05].
- The targeted market.
- The place where the data is stored.

---

2 Adapted from the figure of ‘Shing Hin Yeung’. https://commons.wikimedia.org/wiki/File:ERP_Modules.png
Based on where the data is stored and how the system is managed, ERP systems can be categorized into three types. These types are [Gro12, CDHM14]:

**On-Premise ERP**

In this type, everything is stored and installed locally in the organization infrastructure. It is the organization responsibility to manage the infrastructure and make sure that the system is always up-and-running [Gro12]. This type is suitable for large size organizations.

The deployment of On-Premise ERP needs time, and usually, the cost to start the system is high. However, On-Premise ERP long term costs will be less because it is limited to the maintenance, and upgrade costs. This type of ERP can be easily customized, and the organization has full control over the ERP system and its data.

**Cloud ERP**

Cloud ERP is more suitable for small and medium size organizations. The ERP system and its data is managed by a hosting company. The organization’s user can access the ERP system via internet [Gro12], which allows the users to have real-time access to the ERP system using computer browsers or mobile devices. Cloud ERP is faster and cheaper to be implemented. Furthermore, it requires fewer Information Technology (IT) specialist because the system is managed and maintained by a hosting company. One disadvantage of Cloud ERP that, it is less flexible than On-Premise ERP system. The main advantages of Cloud ERP compare to On-Premise ERP are that it is easily scalable for adding more users and easily upgraded.

**Hybrid**

Hybrid ERP system combines both On-Premise and Cloud ERP systems [CDHM14]. The main idea of Hybrid ERP systems is having some ERP modules that require real-time access or mobility, such as e-commerce, in the cloud and other modules can stay as On-Premise ERP modules, such as inventory management.

The selection of the right ERP system type for a specific organization is controlled by different factors, for example:

1. The organization size (small, medium, large).
2. The organization’s budget for managing ERP system.
3. The IT expertise within the organization.

For instance, a small organization that cannot afford to spend a lot of its budget on implementing an ERP system should use Cloud ERP. On the other hand, Implementing On-Premise ERP system in a big organization that has a good IT team would be more flexible and efficient.
2.1.2 Available ERP Systems

Many ERP systems are available by different companies. As shown in Figure 2.2, which was generated based on data collected from the Panorama Consulting website in the period (06-2014 to 10-2015) [Sol15], there are few companies (SAP, Oracle, Infor and Microsoft) that dominate the market with around 64% ERP systems market share in 2015, leaving 38% market share for all other ERP system companies. Some of these companies, such as Odoo (OpenERP previously), are developing an Open-source Software (OSS).

![Market Share Chart]

Figure 2.2: ERP Vendors

[Sol15]

In the next two sections, we define the differences between the proprietary software and the open source systems.

Proprietary Software

Proprietary Software or Commercial Software can be defined as “computer software licensed under the exclusive legal rights of the copyright holder.” [PK13].

The usage of this type of software is determined by the contract between the user and the owner called End-User License Agreements (EULA) [PK13]. The user should accept the contract conditions before buying the software license, normally, the user does not have the right to copy or re-distribute the software without the acceptance of the software owner [PK13]. Any violation to the terms of use by any side can have legal consequences.
2.1. Enterprise Resource Planning Systems

The source code of Propriety software normally is not available for the user. Furthermore, the user does not have the right to modify, copy or re-distribute the software [PK13]. Any changes to the software should be done by the vendor.

The cost of Propriety software is high and the user may pay annual license renewal [PK13], but this high cost comes with quality support and more stable software [Inf15]. Another drawback of Propriety Software is that the user depends totally on the software owner and has little influence on the system new updates [Inf15].

**SAP ERP** is a proprietary ERP developed by Systems, Applications & Products (SAP) SE company located in Germany. SAP is considered the leading in the ERP system industry\(^3\), SAP provides three deployment options for ERP system (On-premise, cloud, hybrid) and it supports companies of different sizes. SAP ERP is developed on top of SAP HANA\(^4\) database. SAP HANA is in-memory, Atomicity, Consistency, Isolation, Durability (ACID) compliance database developed by SAP which is suitable for both Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) data processing models.

**Open-source Software (OSS)**

Open source systems are systems which their source code are available for users, and the users are free to modify the source code based on their needs and redistribute the system [PK13].

In OSS, the user can update the source code to customize the software. However, these updates should be added carefully by expert. Otherwise, it may lead to software’s bugs. On the other hand, in Propriety software, the users should ask and pay the company that developed the system for modifications.

The existence of large community of developers and testers for open source systems can result in several technical benefits [MF]:

- **Security**: The accessibility for the source code increases the awareness of the security flaws in the system.
- **Flexibility and freedom**: By allowing the users to modify and customize the system freely.
- **Lower cost**: The user does not have to pay license fees to use the system [PK13].
- **Quality**: “Given enough eyeballs, all bugs are shallow.” (Linus Torvalds\(^5\)) [TKK14].
- **Innovative**: Since the source code is available that will give a motivation for developers to be creative and develop new ideas.

---

\(^3\)[http://go.sap.com/product/enterprise-management/erp.html]

\(^4\)[http://go.sap.com/product/technology-platform/hana.html]

\(^5\)The creator and, for a long time, principal developer, of the Linux kernel.
Also there are several drawbacks of the open source systems, such as:

- No system guarantee: Most OSSs do not have company that supports the system in case of problem. They depend on the supported community [PK13].
- Insufficient documentations: Not all OSSs have a proper up to date documenta-
- Access the source code may lead to problems if not updated professionally [MF].

2.2 Odoo

Odoo is an open source ERP system known previously as OpenERP. Odoo is con-
- considered the highest installed business application worldwide with more that 2,000,000 users [Odoo]. Odoo offers both On-Premise and Cloud ERP system. It consists of 30 main applications such as (sales, e-commerce, invoicing, accounting and user website management). In addition, more modules and applications have been published by de-
- In the Figure 2.3, we present the main ERP modules that the user can select when subscribes to in cloud version of Odoo.

![Odoo On Cloud Modules](https://www.odoo.com/apps/modules)

Figure 2.3: Odoo On Cloud Modules


[7]This figure is based on: https://www.odoo.com/trial
Odoo is developed using Python\(^8\). Odoo provides a standardized way for developers to develop new Odoo modules or customize and modify already existed modules. Odoo modules consist of several models which interact with each other’s and with other modules to achieve the goal of the developed module.

**Model inheritance** and **View inheritance** are the main features in Odoo which allow the developer to add new features to a model or view and modify an existing model or view.

There are three types of models in Odoo. Each one of these types consists of *attributes* which reflect the model state, and *functions* which determine the model actions. In addition, the model contains a set of predefined attributes that have a specific meaning for Odoo:

- **Normal model:** This model represents a real business object. For each model, there is a database table. This table stores data related to the instances of a normal model. Odoo ORM is responsible for creating the appropriate table for a model and performing the database operations.

- **Abstract model:** When there are several models that share common functionalities instead of re-writing the same code several times, the developer can create an abstract model containing these features. All other models can inherit these features from it.

- **Transient model:** This model is similar to Normal model in having a database table. Nevertheless, the data stored by this model is only available temporally. This model is useful for Graphical User Interface (GUI) development.

Odoo ORM layer is a middle-ware component that facilitates the communication between Odoo and the relational data storage system to perform database Create, Read, Update, Delete (CRUD) operations. As shown in Figure 2.4, Odoo

![Figure 2.4: Odoo ORM](image)

In general, the purpose of the ORM layer is to provide the system with a flexibility to use different database storage systems without breaking the system. This flexibility is

\(^8\)https://www.python.org/
gained by isolating the business objects from their persistence storage [DR04]. Nevertheless, Odoo ORM supports only PostgreSQL as a database system. To enable Odoo to communicate with other data storage systems requires a lot of modifications.

Mail messages and attachments are two special Odoo modules because these two modules can be used by any other module in Odoo, such as ‘product.product’ and ‘res.users’ modules. Further, attachments and messages objects can be linked to any business object inside Odoo. Based on these properties, during the lifetime of the system the information stored in these two modules becomes huge, which reflects in a bad performance of the overall system.

2.2.1 Mail Messages Module:

The messages module manages the messages inside Odoo. These messages can be created by other Odoo modules and linked to business models [Odob].

In Figure 2.5, we illustrate the physical Entity Relationship Diagram (ERD) for the mail message module and show the main columns in ‘mail_message’ table:

- **ID**: Auto increment integer to identify the message.
- **Create Date**: Date of inserting the message.
- **Write Date**: Date of last update of the message.
- **Create UID, Write UID**: The ID of the user who created and updated the message respectively.
- **Body**: The text of the message.
- **Model**: The name of a model that created the message.
- **Res ID**: The id of the business object this message linked to.
- **Record Name**: The name of the record this message linked to.
- **Type**: The type of the message: notification, comment, ...etc.
Figure 2.5: Mail Message ERD
In Table 2.1, we show an example of the data stored in the mail messages table: The first row is a message that has been sent to Odoo users after installing Human Resource (HR) Odoo application via ‘mail.group’ module. The ID of this mail group is ‘1’. For each user, a record will be inserted in ‘mail.notification’ table to notify the users to check this message. The second row is a notification message that is linked to an object of ‘product.product’ module with id ‘2’. To keep track of the product life cycle, this message should be displayed whenever the product with id ‘2’ is viewed.

<table>
<thead>
<tr>
<th>ID</th>
<th>Body</th>
<th>Model</th>
<th>Res ID</th>
<th>Reccond_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Manage your human resources with Odoo.....</td>
<td>mail.group</td>
<td>1</td>
<td>Whole Company</td>
</tr>
<tr>
<td>106</td>
<td>Product created</td>
<td>product.product</td>
<td>2</td>
<td>Service</td>
</tr>
</tbody>
</table>

### 2.2.2 Attachment Module:

This module is responsible for managing the attachments that may be attached to any other Odoo module objects [Odob]. Attachments can be added as binary files or store them on local file system and save the location in the database.

In Figure 2.6 on the next page, we show the physical ERD for attachments module. ‘ir.attachment’ is the table where attachments’ information is stored. The relation between the ‘mail_message’ and ‘ir.attachment’ is many-to-many relationship represented by the table ‘_message_attachment_rel’.

The main columns in ‘ir.attachment’ table are:

- **ID**: Auto increment integer to identify the attachment.
- **Create Date**: Date of inserting the attachment.
- **Write Date**: Date of last update of the attachment.
- **Create UID, Write UID**: The ID of the user who created and updated the attachment respectively.
- **Type**: The type of the attachment: Url, Binary.
- **Mimetype**: The file type of the attachment: JPEG, PDF...etc.
- **Res name**: The name of the attachment.
- **Res model**: The module that this attachment linked to.
- **Res id**: The id of the module’s object.
- **Datas big**: Binary field that contains the attachment if the type in binary.
Figure 2.6: Attachment Module ERD
Background

- **Url:** The Uniform Resource Locator (URL) indicates where the file is stored when the attachment type is Url.

Same as the mail messages module, the combination (model, res_id) links the attachment with any business object from any module in Odoo.

### 2.2.3 Usage of Mail Message and Attachment Modules:

In this section, we illustrate how the mail messages and attachments modules can be integrated with any Odoo module.

To clarify that, we explain one of core Odoo modules which is Product module. Product module is integrated with mail messages and attachments modules. In Listing 2.1, we present the eXtensible Markup Language (XML) code for the view page of a product. The code between the ‘div’ tag is rendered into a message widget viewed in Figure 2.7. This code can be used by any other Odoo module to link it with the mail messages and attachments modules.

```xml
[....]
<!-- base structure of product.template, common with product.product -->
<record id="product_template_form_view" model="ir.ui.view">
  <field name="name">product.template.common.form</field>
  <field name="model">product.template</field>
  <field name="mode">primary</field>
  <field name="arch" type="xml">
    <form string="Product">
      [....]
      <div class="oe_chatter">
        <field name="message_follower_ids" widget="mail_followers"/>
        <field name="message_ids" widget="mail_thread"/>
      </div>
    </form>
  
  <div>
    </field>
  </record>
[....]
```

**Listing 2.1: Integrating Mail Message with Product**

In Figure 2.7, we present the view page of a product. First, the product’s information is listed. Then, at the bottom of the page, the mail messages widget is shown. This widget presents the messages that are linked to this product and allows any Odoo user to post a new message. In addition, this widget displays the auto-notification messages that has been generated by Odoo.

The messages in Figure 2.7 are displayed in create date descending order. The first message is created by the administrator, and it contains a picture of the product. This

---

9Peice of Code from Odoo core file “product_view.xml”
picture is stored as attachment related to the containing message and to the selected product. The second message is an Odoo auto-notification message, which indicates the creation of this new product.

![Figure 2.7: Product View](image)

In Figure 2.8, we display a snapshot of the database table “mail_message” after creating the above product.

For each product, Odoo also creates a product template object. The three messages are linked to the product or the product template by the combination (model, res_id) which actual values are (product.product, 15) and (product.template, 15) respectively; that refers to two objects (product.product, product.template) and these objects’ identifiers are (15, 15).

![Figure 2.8: Mail Message Data for New Product](image)

---

10Screenshot of view a product page from Odoo system
2.3 NoSQL Database Storage

The Web 2.0 applications have changed the way the web is used. It is normal these days for any web application to serve millions of users simultaneously. These users generate high number of requests and enormous amount of data, which the data storage system needs to manage efficiently to provide a good service for the users. Many big web companies such as Google, Facebook, Twitter and Amazon have noticed the limitation of relation databases to support such requirements even with powerful computers [V14]. To overcome the problem many companies developed own data storage system that supports horizontal scalability to adapt for the new requirements, these systems later become known as NoSQL data storage which can be interpreted as “Not Only SQL” or “Non Relation” [Cat11]. There is no specific definition for NoSQL data storage but most of these new systems share specific characteristics [Fow12, MH13]:

- NoSQL data storage systems do not use tables as basic building blocks to store their data. There is no predefined schema, each data item can have different attributes.
- They are distributed systems, developed to support parallel processing for huge data volumes.
- NoSQL systems do not necessarily support accessing the data via SQL statement, each of them has different query language.

In the following sections, we describe NoSQL data storage in more details and provide a comparison between traditional SQL and NoSQL data storage systems.

2.3.1 BASE Characteristics for NoSQL

NoSQL focuses on the performance, which can be gained by data replication, distribution and the ability to store unstructured and semi-structured data. But this performance improvement comes with lack of consistency, which is the focus for SQL data storage systems [ABF14].

RDBMSs constrain their operations to fulfill the ACID properties:

- **Atomicity**: Either all operations in the transaction must be executed or none of them should be performed.
- **Consistency**: Transactions transfer the database from a consistent state into another consistent, eventually changed state.
- **Isolation**: Transactions are executed independently. No transaction has access to uncommitted state of another transaction.
- **Durability**: Once the transaction has been committed the changes must be persistent.
The full consistency approach that RDBMSs follow by conforming with ACID properties has a bad impact on the performance of the database systems [FK09]. However, most of web-based applications do not need this level of consistency [BFG+08], while the availability and the scalability are required. In contrast to SQL systems, NoSQL systems are developed based on BASE properties which favor the availability over the consistency [ano11]:

- **Basically available:** Lower the probability of data becomes unavailable through replicating and partitioning the data across several servers to return a subset of data even in case of failure of some nodes. The replicating and the partitioning of data allow NoSQL to response to a high number of operations per second.

- **Soft state:** The state of the system may change even if there is no input, in order for the system to be consistent in all nodes and replicas.

- **Eventually consistent:** NoSQL system ensures that in a point in the future the system will be in a consistent state.

**CAP Theorem**

Distributed applications are controlled by the CAP theorem by Eric Brewer. The three components of CAP theorem are: **Consistency, Availability and Partition Tolerance**.

According to CAP, when working with distributed systems such as: Distributed web services and distributed database only two components out of the three can be accomplished [ABF14].

In centralized relational database, **Partition Tolerance** is not fulfilled so this database can be strictly consistent and available as long as the node is available. The majority of NoSQL data storage systems are developed to be distributed so **Partition Tolerance** should always be fulfilled. Thus, the system needs to choose between **consistency** and **availability**. That does not mean that this decision is a sharp decision either the system is consistent or available, but the relation between these two factors can be a trade off relation. The more consistency is required the less availability the system has and vice versa. It is important to note that most of the time the system tradeoff the consistency with the response time instead of availability [Fow12]. In some domains, such as finance applications, the consistency is more important than availability but in other domains such as e-commerce systems the availability is the most essential.

There are a lot of data storage systems available these days, it is the organization responsibility to select the best of these systems.

In Figure 2.9, we show CAP triangle and illustrate most common NoSQL data storage in accordance to the CAP theorem.

- **CA:** This category does not support data partitioning, cares more about consistency and availability such as: RDBMSs, Aster Data. The system is available as long as the server running the database is available.
2. Background

Figure 2.9: CAP Theorem

- **CP**: In this category, data is partitioned across nodes, and the consistency is preferred to the availability such as: Google’s BigTable\(^{11}\), Apache’s Hadoop Database (HBase) and MongoDB\(^{12}\). Database systems in this category return always correct, up to date results but with higher response time.

- **AP**: Same as previous category, data is partitioned across nodes, but this category favors the availability over the consistency such as: Amazon’s Dynamo\(^{13}\), Facebook’s Cassandra\(^{14}\) and CouchDB\(^{15}\). Returns the results fast but sometimes may return outdated values.

### 2.3.2 SQL v.s. NoSQL

NoSQL systems have been developed to overcome SQL systems limitations in managing and processing the big data which is produced by the existing technologies. In this section, we compare between common features in NoSQL databases and the SQL databases.

\(^{11}\text{https://cloud.google.com/bigtable/}\)
\(^{12}\text{https://www.mongodb.com/}\)
\(^{13}\text{https://aws.amazon.com/dynamodb/}\)
\(^{14}\text{http://cassandra.apache.org/}\)
\(^{15}\text{http://couchdb.apache.org/}\)
2.3. NoSQL Database Storage

SQL databases

**Structured data:** SQL systems store data in predefined schema. First, the tables of the database should be created to define the schema. Then, data can be stored. All data items in a table have the same attributes.

**Scale up:** To increase the performance of SQL systems, the user needs to run it on more powerful server. Although SQL systems can run on distributed environment, it was designed to run on a centralized environment.

**Impedance mismatch problem:** The development using SQL systems has the “impedance mismatch” problem, where the storage structure is different from the in-memory structure and the GUI presentation. That forces the developers to update several tables and make joins to store or present related data. Developers try to solve this problem by using Object Relation Mapper, such as Odoo ORM.

**Structured Query Language:** SQL systems support standard language to access stored data, once the user knows SQL commands, the user can manipulate the data of majority of SQL databases.

**ACID properties:** SQL systems insure that the user will always read the most recent value, even if that leads to lower performance. Which makes SQL systems the best choice for the domains that concern about data integrity, such as Banking systems.

**Stable:** SQL databases have been in use for years and they are stable.

NoSQL databases

**Unstructured data:** NoSQL systems can store structured, unstructured and semi-structured data. Data items can have different attributes from each other’s. That solves the problem of having “Spare data set” with many null values.

**Scale out:** NoSQL systems are developed to run on clusters of commodity and distributed hardware. To increase the performance, we can add more nodes to the cluster.

**Aggregated data:** NoSQL systems can store aggregated data. “impedance mismatch” problem can be solved by storing the related data together. For instance, data which is being queried or updated together often, can be stored in the same document or in two different column families within the same row.

**Different query languages:** Since most of NoSQL systems have been developed by individual companies and for different purposes, each of these systems has own language to access stored data.

**BASE properties:** NoSQL systems follow the rule that data will be eventually consistent. Thus, sometimes the users may read outdated information to gain better performance. So, some businesses prefer NoSQL systems to give the users more satisfiable services, such as E-commerce systems.

**Evolving:** NoSQL systems are still in the growing phase, and continuously there are new releases for different projects.
SQL and NoSQL database systems are not replacements for each other, but each of them is useful for different use cases [NLIH13].

### 2.3.3 NoSQL Data Model

NoSQL database systems have been developed to solve different data issues, these systems can be categorized based on their data model into four categories [MH13]:

- Key-Value stores.
- Document databases.
- Graph stores.
- Column-Family databases.

In the following sections, we explain the data models which are used in each of these categories.

#### 2.3.3.1 Key-Value Stores

The base for all NoSQL database systems. The data model in this category stores data as key-value pairs, it is similar to Hash Table data structure but data is stored on persistent storage [SD12]. The values can be of different types: simple text, JavaScript Object Notation (JSON) objects, images or any complex binary object. This data model is useful for fast and scalable retrieval of data based on the key only. Amazon's Dynamo is an example of this data model where it is being used in Amazon’s shopping cart and customers’ session data [NLIH13].

In Figure 2.10, we show how may the mail message object be stored in a Key-Value database. Even though this storage type improves retrieving the mail message data by mail message ID, it is not suited in our use case. The problem of this data model that we can not search the message object based on different attributes such as “res_id” and “model”.

<table>
<thead>
<tr>
<th>Message</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>{model: res.partner, type: notification, body: &lt;p&gt;Partner Created.&lt;/p&gt; res_id: 20, ...}</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>{model: library.book, type: comment, body: &lt;p&gt;The Book is sold.&lt;/p&gt; res_id: 31, ...}</td>
</tr>
</tbody>
</table>

Figure 2.10: Key-Value Data Model
2.3.3.2 Document Databases

As the name implies, this set of NoSQL databases stores data items as documents which are hierarchical key-value pairs. These documents can be represented in a semi-structure language, for example: XML, Yet Another Markup Language (YAML) and JSON [SM12]. Each document can have own attributes that may differ from others. Users can search by any attribute values, retrieve a document or part of it. Besides, the user can update any part of a document. This data model is good for storing [SM12, TPP13]:

- Data that can be represented as documents.
- Aggregated data.
- Data that contains items with different attributes.

An example of this data model is MongoDB database system.

Listing 2.2 illustrates how we can store the mail message and its related attachments in a document oriented database.

```json
{
  "id": 1,
  "model": "res.partner",
  "type": "notification",
  "res_id": 31,
  "attachments": [{"id": 4,
    "name": "ProfilePicture.png",
    "datas": <BLOB>,....
  },{....
  },{....
  }]
}
```

Listing 2.2: Document Oriented Databases

This data model can be used to store mail messages and attachments data but the problem is its query language. For example, in MongoDB to retrieve the title of the message with ID equals '3':

```python
db.message.find({id: 3}, {title: 1})
```

This query depends on Python function calls and the query parameters should be passed as documents. Whereas, Odoo ORM generates dynamic strings that represent SQL queries. Then, these queries are executed on the database system. Updating Odoo ORM to support such document oriented queries requires major changes in the ORM.
2.3.3.3 Graph Stores

This special type data model can be used when the relations between data items are more important than the data item itself. The data in this model is represented as a graph. The nodes are the data items and the edges are the relations between these items. There are four types of graphs: Simple graphs, attributed graphs, nested graphs and hyper graphs [TPP13].

Obvious examples for using this data model are social networking applications, such as Twitter and Facebook. These applications are interested in relations between people and how they may interact with each other’s. This data model comes with a query language that can explore the relations smoothly and quickly instead of writing complex queries with many joins, which tend to have a bad performance [YT14]. Examples of this database types, InfoGrid\textsuperscript{16} and Neo4J\textsuperscript{17}.

In Figure 2.11, we show a representation of how mail messages and attachments may be stored using Graph store data model. The nodes are the model instances and the edges are the relations between these models. For example, ‘User1’ (instance of ‘res.users’) posted ‘Message1’ (instance of ‘mail.message’) which has two attachments ‘attach1’ and ‘attach2’ (instances of ‘ir.attachments’). Furthermore, ‘Message1’ was posted in the view page of ‘Product1’ (instance of ‘product.product’). Properties of the any instance can be store within the node information.

\textbf{Figure 2.11: Graph representation for Mail messages and attachments modules}

\textbf{NoSQL} systems which use this data model are not suitable for our use case. To use such systems, more Odoo modules which are related to mail messages and attachments modules (such as ‘res.users’) should be stored in this new system. Moreover, the query

\textsuperscript{16}http://www.infogrid.org/
\textsuperscript{17}https://www.neo4j.com/
language of this data model does not satisfy our needs. It is a query language that concerns more about the relations between the data items, whereas we are interested in the data item itself.

### 2.3.3.4 Column-Family Databases

This data model stores the data in a column-oriented manner, in which data of a column is stored on the disk consequentially, instead of data of a row [Geo11]. Each data item is identified by a row-key and has a set of attributes. These attributes are stored as Key-Value pairs, the key can be called column. Each key can have several versions of a value with attached timestamp which shows when this value was stored in the database. Each set of related columns forms a column family. The table consists of one or more column families [TPP13]. The rows which are stored in a column family can have different columns, depends on each row needs.

In column family databases, we can store the related information together in the same row using different column families. For example, instead of storing the order and its items in two tables we can store them in one row within different column families. This aggregation is useful especially when the data is distributed across several nodes in the cluster [Fow12]. An example of this data model is Google’s BigTable. BigTable runs on top of Google Distributed File System (GFS). Several other database systems have been inspired from BigTable, such as HBase which can run over Hadoop Distributed File System (HDFS) [TPP13].

We illustrate this data model in Figure 2.12. We show how three rows consist of two column families may be stored in the column-oriented data model.

![Figure 2.12: Column-family Data Model](Geo11)

The main advantage of this data model is that it manages distributed and replicated data efficiently [TPP13]. Furthermore, this data model supports large-scale, parallel data processing frameworks, such as Hadoop MapReduce with HBase [Geo11].
In this thesis, we use HBase and Hadoop as a NoSQL data storage. In Chapter 3, we explain what are the advantages of this selection and how mail messages and attachments can be modeled and stored.

Finally, the existence of several database storage systems makes it difficult to decide which one is the best option for a specific application; the answer of this question depends on how the organizations use data and what kind of operations will be performed.

2.4 Hadoop Ecosystem

Hadoop ecosystem is a set of mostly open source applications offered by Apache Software Foundation for processing large data sets. These applications were developed to utilize Hadoop framework and MapReduce algorithm to provide distributed data processing [MMI13]. As examples of these applications:

- Hadoop: The main component of the ecosystem\(^{18}\).
- HBase: Distributed, column-oriented database storage system\(^{19}\).
- Spark: For fast data analysis, performs in-memory MapReduce jobs\(^{20}\).
- Pig: Powerful scripting language to express data flow\(^{21}\).
- Sqoop: SQL to Hadoop, transfers the data from SQL data storage to Hadoop and vice verse\(^{22}\).
- Hive: Support SQL queries on large data stored in HDFS, by translating each query into a MapReduce jobs\(^{23}\).

The organization can built its own stack of Hadoop ecosystem applications in order to solve its data problem. To cut down the time and the effort to implement Hadoop and integrate the different applications, organization can use commercial frameworks such as: Cloudera\(^{24}\), Hortonworks\(^{25}\) and MapR\(^{26}\), which provide an integrated distribution of Hadoop and the tools needed to manage it easily.

In the next chapters, we explain in details the parts of Hadoop ecosystem that we use in this thesis and illustrate the benefits of each one of them.

\(^{18}\)http://hadoop.apache.org /
\(^{19}\)http://hbase.apache.org/
\(^{20}\)http://spark.apache.org/
\(^{21}\)http://pig.apache.org/
\(^{22}\)http://sqoop.apache.org/
\(^{23}\)http://hive.apache.org/
\(^{24}\)http://www.cloudera.com/
\(^{25}\)http://hortonworks.com/
\(^{26}\)https://www.mapr.com/
2.4.1 Hadoop

Hadoop is a programming framework built using Java. Hadoop is designed to execute distributed data processing over commodity hardware. It enables the developers, even without any distributed or parallel programming skills, to write distributed data processing jobs.

Hadoop consists of three main components [Fou]:

- Hadoop Common
- Hadoop Distributed File System
- MapReduce Framework
- Yet Another Resource Negotiator (YARN), since Hadoop version 2.0 [Whi12].

Hadoop common is an essential module of Hadoop project which consists of set of utilities which used by other Hadoop’s modules. It contains components that support I/O operations and help the HDFS [Fou, Whi12].

2.4.1.1 Hadoop Distributed File System

HDFS is the key component of Hadoop. It manages huge data set reliably. Data can be stored and replicated in a cluster of commodity servers and performs quick, sequential access to this data [SKRC10]. Furthermore, HDFS is designed to perform fast read operations but the write operations are slower, so HDFS is optimal for dealing with data that has the access pattern write-once/read-many [SRC10].

HDFS partitions the files into blocks and distributes these blocks among the cluster’s nodes also it replicates the blocks across the nodes [SRC10]. The block size which is the finest data unit that HDFS works with. The replication number of the blocks is easily configurable. In Hadoop cluster, there are two types of nodes [SKRC10, Whi12]:

- **Namenode**: The master node. Namenode stores the meta-data of the HDFS which is called HDFS namespace. HDFS namespace consists of the hierarchical structure of directories and files which are stored in the HDFS. Also, it identifies the datanodes that are responsible for storing each block. If any client needs to write or read data, first, the client asks the namenode to return the datanodes that should be communicated to store or retrieve the data. Before Hadoop 2.0, namenode was a single point of failure. Then, a new backup namenode has been added which is called “Secondarynamenode”.

- **Datanode**: The slave node. Datanode stores the actual data of the files that are existed in HDFS. The datanode stores two files: One contains the data and the other is checksum file to identify corrupted data blocks.
HDFS was designed to expect the hardware failure as normal event. To detect the hardware failures, HDFS uses heartbeat technique [SKRC10]. Each datanode regularly sends a heartbeat signal to the namenode. Once the namenode detects missing heartbeats from any datanode, it marks this datanode as out of service and re-replicates all data that is stored on this datanode to other datanodes to satisfy the minimum replication factor.

### 2.4.1.2 MapReduce Framework

MapReduce is a programming model for processing large and distributed data [DG08]. Originally, it was introduced by Google to solve the problem of indexing the crawled web pages. MapReduce is built based on the “Data Locality” concept which means moving the computation to data instead of moving data to the computation. So the processing algorithm runs on each node that contains the data and then aggregates the intermediate results to get the final output [DG08].

MapReduce consists of three main steps:

- **Map**: This step transfers the input data into set of \(<key, value>\) pairs, map function performs on the node where data exists.

- **Shuffle and Sort**: In this step, Hadoop shuffles the intermediate \(<key, value>\) pairs to one or several reducers and sorts these pairs on the key.

- **Reduce**: The final step which applies the required functions, such as (count, average, any custom grouping functions), on the intermediate results. Then, it writes the final output to HDFS.

To run a MapReduce job, first, the job should be submitted to the **Job Tracker** which is a service that normally runs on the namenode. This service splits the MapReduce job into several Map processes and one or several Reduce processes. The actual Map and Reduce jobs are managed by the **Task Tracker** which runs on each datanode. Once the task trackers finish their jobs, they notify the job tracker to mark the job as finished [Whi12].

Let us assume that, we need to use MapReduce job to count the words frequency in the following set of words: “Odoo, ERP, Odoo, NoSQL, NoSQL”.

This set is stored in a file in HDFS cluster of three nodes with a replication factor equals to ‘2’.

In Figure 2.13, we present the results of each step of the MapReduce job.

First of all, we need to define our Map and Reduce functions. Then, the job should be submitted to the job tracker which splits the job into three map programs. Each one of these mappers runs on a portion of locally stored data in a different datanode. Each map process is managed by the task tracker on the node which the process runs on. When the mappers finish their jobs, Hadoop framework moves the intermediate results
2.4. Hadoop Ecosystem

2.4.1.3 Hadoop YARN

YARN is a resource management framework to assign cluster’s resources to different applications; it was introduced in Hadoop v2.0 to decouple the resource management from the MapReduce framework [Whi12]. YARN allows running various type of applications on a Hadoop cluster in addition to MapReduce jobs without affecting each other.

YARN also has a Master/Slave architecture [Whi12]:

- Master: It is the YARN Resource Manager. It coordinates the resources (such as: Central Processing Unit (CPU)s and Random Access Memory (RAM)s) usage between the slaves.

- Slave: Each application that runs on the Hadoop cluster should have an Application Master. Application master negotiates with the resource manager to acquire the needed resources to start the application tasks. Then, it releases the resources when the application finished its tasks.

2.4.2 Zookeeper

In a distributed environment, the coordination and synchronization between the system’s components are essential services to guarantee that the system works correctly. Zookeeper offers these services for Hadoop ecosystem’s applications.

Zookeeper is an open source project developed by Apache to coordinate distributed services. Zookeeper provides many services, for example [Whi12]:

- Preserves the configuration information.

- Manages distributed synchronization, such as locks and timestamp.
• Provides group services: Replicates data after node lost, elects leader ...etc.

• Maintains the list of active nodes.

Zookeeper data model consists of several nodes called *zNodes*. These nodes store the applications’ data (hostnames, locks, general configuration) and can act as containers of other znode. Zookeeper insures high availability for data stored in its znodes and guarantees atomic, reliable and sequential updates to return the same and the newest information to the requesting processes\(^\text{27}\).

### 2.4.3 HBase

Hadoop database is a distributed, column-oriented *NoSQL* database system which can run on local file systems as well as the HDFS file system [Geo11]. HBase supports managing large tables with billions of rows effectively and utilizes MapReduce framework to provide batch data processing.

HBase does not support SQL to access the data, instead it uses its own query language with predefined filters that written like Java function calls. For example, the following SQL query:

```sql
SELECT TITLE, BODY
FROM MAIL.MESSAGE
WHERE MODEL = 'RES.PARTNER'
```

In HBase, the equivalent query can be written as:

```java
SCAN 'MAIL.MESSAGE',
COLUMNS => {'COLFAM:TITLE','COLFAM:BODY'},
FILTER =>{
       "SingleColumnValueFilter('COLFAM','MODEL','=','binary:RES.PARTNER')"
}
```

Although HBase does not support SQL queries, there are applications in Hadoop ecosystem that add an SQL layer over HBase, such as Phoenix.

In Figure 2.12 in Section 2.3.3, we illustrate the column-family data model which is HBase data model [Geo11].

• The table consists of several column families. The column families need to be defined in the time of table creation.

• The row key is important in HBase, because HBase depends on it to determine in which node this row should be stored or replicated.

• Each row can have different columns in each column family.

\(^\text{27}\)https://zookeeper.apache.org/doc/trunk/zookeeperOver.html
• **HBase** stores versions of each column-value pair and associate this version with a timestamp.

HBase stores its data in files called HFile. If HBase runs over HDFS, these files are managed by HDFS otherwise by a local file system. HBase uses Zookeeper to manage the coordination and synchronization between its nodes. There are two types of nodes [Geo11]:

• **HMasterServer**: The master node which stores meta-data information, assigns regions to regionservers, and detects a node failure.

• **RegionServer**: In HBase cluster, there are several slave nodes which are called regionservers. Regionserver responsible for managing the CRUD operations on the regions’ data. Region is the actual container of data and consists of subgroup of table’s rows.

2.4.4 Phoenix

Apache defines Phoenix as “We put the SQL back in NoSQL”\(^{28}\).

Phoenix is a relational layer over the HBase database, it provides an SQL interface to access HBase data. Phoenix executes the SQL query by breaking the query into several HBase scan processes. Then, Phoenix performs these processes in parallel across the HBase cluster [Hai16, Tay15]. Phoenix accesses the data using native HBase Application Programming Interface (API) instead of MapReduce jobs [Hai16]. Phoenix maps each HBase table with Phoenix table and stores meta-data about that table in order to access its rows.

In the time of writing this thesis, the last stable version of Phoenix is 4.8. This version supports many functionalities of RDBMSs, for example [Fou16]:

• Transactions.

• Secondary and functional indexes.

• “Order by” and “Limit” statements.

• Aggregation functions (sum, avg, min, max).

Furthermore, Phoenix can be integrated with other Hadoop ecosystem applications such as Spark and it can perform MapReduce jobs.

\(^{28}\)https://phoenix.apache.org/index.html
3. Design and Implementation

In this chapter, we introduce our solution to optimize Odoo latency which is caused by mail messages and attachments modules by enabling Odoo to communicate with HBase as a data storage for these two modules. In Section 2.4, we explain the Hadoop stack that we use in our approach. After that, in the Section 3.3, we explain in details how we enable Odoo to change its data storage from PostgreSQL to HBase. Finally, in the Section 3.4, we provide an overview about our solution architecture and explain how system components interact with each other’s.

3.1 Requirement Analysis

In this section, we explain the requirements of our developed solution. Functional requirements describe the main functionalities that the system must provide, while the non-function requirements control how these functionalities are performed.

Functional Requirements

To optimize Odoo performance, we need to adjust Odoo to store mail messages and attachments data into NoSQL data storage in order to get the benefits of a scalable data management system. So, the main functional requirements are:

- Implement all Odoo functionalities, such as: Add, update, read and search for mail messages and attachments modules to support the new data storage.

Non-Function Requirements

Odoo is a web-based ERP system, the process time that is needed to return or update the data is highly noticed by the system’s users. Our main non-function requirement is to minimizing the process time which is essential for a better user experience.
The correctness of our solution should be compared to BASE properties instead of ACID properties. That means, the result of the system can be considered correct even if the result is outdated. However, once the changes are propagated to all the nodes in the cluster, the system must return the newest values. This is valid for our use case mail messages and attachments modules, because it is not important to display the newest data immediately. On the contrary, this consistency level is not acceptable for other Odoo modules such as ‘Sales’ and ‘Purchases’. For instance, two users may buy the same item if both users place the order simultaneously and they communicate with two different data regions.

3.2 Implemented Hadoop Stack

Many open source applications are now parts of Hadoop ecosystem, so before start using the Hadoop ecosystem, we should identify which applications are useful to achieve our objective.

In Figure 3.1, we illustrate the Hadoop stack that we rely on in this thesis:

We use Hadoop version 2.7.2\(^1\) and HBase version 1.2\(^2\) which is compatible with Hadoop 2.7.2.

![Hadoop Stack](image)

Figure 3.1: Hadoop Stack

HBase represents the main data storage system which stores mail messages and attachments data. We configure HBase to used HDFS as its file storage system.

In our approach, we use the HBase built-in Zookeeper instead of stand-alone Zookeeper application.

Odoo is developed to communicate with PostgreSQL database through Odoo ORM layer. Odoo ORM translates the user actions into SQL statements. The main database operations (Create, Read, Update, Delete) are executed directly by calling appropriate Odoo model’s functions (Create, Read, Write, Unlink) respectively. But the search requires dynamic SQL select statements generation to satisfy the users’ needs.

\(^1\)http://hadoop.apache.org/docs/r2.7.2/

\(^2\)https://hbase.apache.org/book.html
In Figure 3.2, we show the steps for performing search operation. First, the user selects the search criteria to filter the data. Then, Odoo compiles this criteria as Odoo domain filter. Domain filter is a list containing the selected filters and the operations between them (such as: and, or, not). Finally, Odoo ORM translates this domain filter into SQL select statement, executes it and returns the results to the user.

![Diagram of Odoo Search Operation Steps](image)

Figure 3.2: Odoo Search Operation Steps

Since HBase does not support SQL statements, to provide such flexibility in our solution, we should choose between two options:

1. Update the ORM layer to compile the domain filter into HBase query language, and executes this query directly on HBase.
2. Use an application from Hadoop ecosystem, such as Hive and Phoenix, that performs SQL queries over HBase.

After analyzing the queries that may be generated by Odoo, we found queries that are not supported by HBase query language and needs further processing to return the expected results. In addition to that, compiling the filter domain into HBase query language will be a time-consuming task, so we choose to continue with the other option which is using an application which provides an SQL interface for HBase.

We compare two open source, Hadoop ecosystem applications that add SQL layer over HBase: Apache Hive and Apache Phoenix. Apache Hive executes the SQL statements as MapReduce jobs that are optimized for high throughput but high latency processing [Whi12]. In contrast to Apache Phoenix which is built to support OLTP with low latency.

In Figure 3.3, we show that HBase-Phoenix is faster than HBase-Hive for performing the query:

```sql
SELECT COUNT(1)
FROM TABLE
```

The table size rang is [10 million, 100 million].

Since we need low latency, we select **Apache Phoenix** to be added to our Hadoop stack.

In this thesis, we use HBase-Hadoop as NoSQL system to store mail messages and attachments, for the following reasons:

---

As discussed in Section 2.3.3, the most suited NoSQL data model for our use case is column-oriented store.

In the Hadoop ecosystem, there are applications that can integrate into HBase to perform SQL queries over HBase.

Using Hadoop ecosystem will enable us, in the future, to develop distributed data analysis using MapReduce framework. Furthermore, we can use any Hadoop ecosystem application.

### 3.3 HBase ORM Odoo Module

“HBaseORM” is our module that facilitates Odoo to store data in HBase. In the upcoming sections, we explain in details our implementation to this module and illustrate the changes that we applied on the normal Odoo data model.

#### 3.3.1 Data Model

To store and manage mail messages and attachments table in HBase, we need to transfer these tables from relational data model to column-oriented data model described earlier.

##### 3.3.1.1 Mail Message

The main required information to define an HBase table is the row-key and the column families.

We use the same auto-increment, integer primary key from original mail message table as row-key for our HBase mail message table, because of the following reasons:
1. Existing data compatibility.

2. Odoo system identifies any model by its integer id, and uses it to navigate through other models.

3. Odoo frequently executes read operation which reads mail message data by its id, and the row key is the fastest way to access the data in HBase.

As shown in Figure 3.4, first, we only move the ‘mail_message’ table into HBase whereas other relational tables stay in PostgreSQL. The mail message table consists of two column families:

**Basic**: It contains the same columns as the original PostgreSQL table.

**HBase**: It contains the columns that are specified only for the HBase table. Basically, this column family is used for de-normalization.

The table ‘mail_message’ has two many-to-many relationships with ‘ir_attachment’ and ‘res_partner’. In relational database, this relation is represented by additional intermediate table which are: ‘message_attachment_rel’ and ‘mail_message_res_partner_rel’,
and to get the linked records with any ‘mail_message’ row, we need to perform a time-consuming join operation.

In HBase, since we can add dynamic number of columns in a column family for each row, we add the id of the linked ‘ir_attachment’ and ‘res_partner’ in the same ‘mail_message’ row. In Table 3.1, we show how to store a message with two related attachments ‘101,102’ and one partner ‘1’.

<table>
<thead>
<tr>
<th>ID</th>
<th>basic...</th>
<th>hbase:partner_ids_1</th>
<th>hbase:attach_ids_1</th>
<th>hbase:attach_ids_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
<td>.......</td>
<td>1</td>
<td>101</td>
<td>102</td>
</tr>
</tbody>
</table>

### Phoenix Mail Message Table

Phoenix is the relational layer over HBase, so it cannot deal with the dynamically added new columns contained in the ‘HBase’ column family. As shown previously in Figure 3.4, Phoenix relation table only covers ‘basic’ column family and does not have access to ‘HBase’ column family. This will be sufficient since we are using Phoenix only for the select SQL statements generated by Odoo. Otherwise, we read directly from HBase.

#### Indexes

Phoenix supports secondary indexes. When a secondary index is created, Phoenix stores meta-data about the indexed column in order to search through it quickly. The most frequently used columns, to filter the message in Odoo are (res_id, model), to accelerate that kinds of queries, we create a complex secondary index on these two columns.

```
CREATE INDEX mail_message_res_model_idx
ON mail_message (res_id,model)
```

Furthermore, all other indexes that were implemented in the relational ‘mail_message’ table, we implement them in our new table. These secondary indexes are on (‘parent_id’, ‘author_id’, ‘subtype_id’).

### Mail Notification

The ‘mail.notification’ model is tightly associated with ‘mail.message’ model. For almost every insert or update operation on ‘mail_message’, there is an insert or update operation on ‘mail_notification’. To reduce access to PostgreSQL when an operation related to ‘mail.message’ model is preformed, we transfer the ‘mail_notification’ table into HBase. ‘mail_notification’ table consists only of one column family ‘basic’ which contains the same columns as the original SQL-table.
3.3.1.2 IR Attachment

The two models ‘mail.message’ and ‘ir.attachment’ share the same usage behavior, but each of them provides different type of information. Every modification we make on ‘mail.message’ table is also applied on the table ‘ir.attachment’.

In Figure 3.5, we illustrate the final data model of both ‘ir.attachment’ and ‘mail.message’ models. We move both tables ‘mail.message’ and ‘ir.attachment’ to HBase, and we link the related rows between the two tables using the dynamically added columns. That allows us to remove the many-to-many relationship table ‘message_attachment_rel’. However, it is not possible to remove the rest of many-to-many relationships tables:

- ‘mail_message_res_partner_rel’
- ‘mail.compose_message_ir_attachments_rel’
- ‘email_template_attachments_rel’.

Because they are used to link the records of PostgreSQL tables with ‘mail_message’ and ‘ir.attachment’.

Figure 3.5: Final HBase Data Model
### 3.3.2 Class Diagram

The class diagram exposes the classes as building blocks in the application and the relations between them.

Our ‘HBase ORM’ module consists of four classes as shown in Figure 3.6. The main class is `HBaseOrm`, which is an Odoo model that implements the communication between Odoo and HBase or Phoenix. We developed this model to be a generalized solution, to create any new Odoo model that it is needed to be managed by HBase. This new model should inherit from ‘HBaseORM’. Then, it can add own customization by overriding the ‘HBaseORM’ functions.

![HBase ORM Class Diagram](image)

Figure 3.6: HBase ORM Class Diagram

‘HBaseORM’ model contains two important data members:

- **hbase_model**: If this property exists in an instance and it is true, Odoo core knows that this is an instance of a HBase managed model.

- **store_in_postgresql**: A boolean property that tells Odoo core if this instance should be stored in PostgreSQL in addition to HBase. If this property is true, all instances of the model will be stored, updated and deleted from both PostgreSQL and HBase, but the data will be filtered and retrieved using PostgreSQL. We added this option to give the Odoo administrator the ability to store everything in PostgreSQL, and also, to perform long running, data intensive reports offline using Hadoop the ecosystem.

‘HBaseORM’ provides implementations for Odoo models’ main functionalities (read, create, write, unlink, search) to communicate with Hadoop ecosystem. In Section 3.3.3 and Section 3.3.4, we explain how these functions work.
MailMessageHbase, MailNotificationHbase and AttachmentHbase apply Odoo inheritance to the basic models mail.message, mail.notification and ir.attachment respectively. This inheritance is defined using the model property ‘_inherit’. Once an inheriting model has been installed into the Odoo system, the functions for the super model will be overridden by the ones defined in the sub-model. This powerful feature allows us to modify the functionalities of the targeted three models.

Although, we used model inheritance feature to modify mail message and attachments models, we had to change the core of Odoo to execute other models tightly related tasks and to remove the hard-coded queries to PostgreSQL tables, that exist inside the Odoo core.

### 3.3.3 Read, Create, Update and Delete

In Odoo, as explained earlier each model has four main functions for CRUD operations. In our implementation for these functions in ‘HBaseORM’, we communicate directly with HBase without the need to Phoenix.

**HBase** is developed using Java and it provides native Java libraries to communicate with it using Java code [Geo11]. Alternatively, HBase provides various Client APIs to access it from other programming languages [Geo11] such as ‘Python’, as in our case. The two main supported Client APIs are: **REST API** and **Thrift API**. We made an evaluation experiment, explained in Section 4.2, on both APIs. As a result of this experiment, Thrift API is faster than Representational State Transfer (REST) API, so in our implementation, we choose Thrift API as the bridge between Odoo and HBase.

There are two Python libraries that we can use to communicate with HBase: Happybase⁴, Starbase⁵. Starbase library does not support Thrift API, it only supports REST API. So, we choose Happybase [Bol] library to send our requests to HBase through Thrift API. Happybase supports all our needs to communicate with HBase, also provides a connection pool to accelerate starting the connection to HBase. Our use case runs Odoo with a PostgreSQL connection pool of size ‘4’. We choose a connection pool of size ‘5’, since mail messages and attachments are queried a lot while a user is navigating through the pages of the system.

### 3.3.3.1 Create

Create function is called by Odoo when a user tries to create a new instance of a model. For the ‘mail.message’ and ‘ir.attachment’ this function is called when a user sends a new message or attach a new file to a specific model respectively.

Figure 3.7 is a sequence diagram that clarifies the create steps for a message object. First, the user inserts new message using Odoo front-end, which triggers the create function for ‘mail.message’ model. We add ‘MailMessageHBase’ model to Odoo. ‘MailMessageHBase’ inherits the basic ‘mail.message’ and our module ‘HBaseORM’. The create function version from ‘HBaseORM’ will be invoked to react to the user’s inputs.

---

⁴[https://pypi.python.org/pypi/happybase/1.0.0](https://pypi.python.org/pypi/happybase/1.0.0)

⁵[https://pypi.python.org/pypi/starbase/](https://pypi.python.org/pypi/starbase/)
‘HBaseORM’ prepares the data to match the new data model and invokes HBase put operation (equivalence to insert in SQL). Using Happybase library and Thrift API. Finally, once HBase returns the response, the user will be informed about the operation’s result.

**Figure 3.7: Create Message Sequence Diagram**

In Listing 3.1, we view a partial peek of the code of the create function implemented in ‘HBaseORM’. First, in ‘HBaseORM’ class, we import our HBase communication library ‘happybase’. Then, we define a connection pool which contains five connections to the master node of our HBase cluster. The signature of the create function, in addition to the mandatory parameter self, has only one important parameter *values*. ‘Values’ is a dictionary that contains all the user inputs and some default values. These default customized values are added by ‘MailMessageHBase’. In our create function, after getting the connection from the connection pool, we iterate through the keys of the values dictionary and generate data that should be stored in HBase table. Finally, we call the ‘baseModel’ create function to complete some Odoo back-end tasks. Then, we return the results to the user. Please, note that we updated the Odoo core to ignore any request related to any HBase model, because we handle these requests.

### 3.3.3.2 Read

Read function takes as input a list of records’ ids and the required information, then, returns JSON string that contains the needed data. The only SQL statement that read function executes is:
Listing 3.1: Odoo Model Create Function

```python
import happybase
pool = happybase.ConnectionPool(
    size=5, host='master', port='9091',
    compat='0.96', transport='framed')

@api.returns(lambda rec: rec.id)
def create(self, values):
    values = self._add_missing_default_values(values)
    [...]
    #Get connection to HBase and store the new values in it
    with pool.connection() as connection:
        table_name = self.map_modelname_to_tablename()
        table = connection.table(table_name)
        [...]
        for field_name in values:
            if (isinstance(self._fields[field_name], fields.Many2many)):
                [...]  
            table.put(self.pack_uint(next_id), information_to_store)
        [...]  
        father = super(HBaseOrm, self).create(values)
    return father
```

SELECT fields
FROM model_table
WHERE ID IN (....)

This is a direct equivalence to returning set of rows from HBase by row key.

In Listing 3.2, we show a piece of read function code in ‘HBaseORM’. The read function signature contains:

- **self**: In Odoo, self is a RecordSet of a specific model which has the records’ ids.
- **fields**: A list of required fields from the database.

First, read function iterates through the RecordSet to pack the requested ids as strings of bytes, and store them in a list. After that, it opens a connection to HBase and executes a batch read to return the required rows by row-key with data of required fields. Finally, the function navigates through HBase results and the required fields to formulate the result as JSON string and return this result to Odoo front-end.
import happybase

pool = happybase.ConnectionPool(
    size=5, host='master', port='9091',
    compat='0.96', transport='framed')

def read(self, fields=None, load='_classic_read'):
    [...] # Prepare data structure for HBase
    for cursor_record in self:
        pack_id = self.pack_uint(cursor_record.id)
        ids_as_list.append(pack_id)
        self_mappers.update({pack_id:cursor_record})
    [...] # Prepare data structure for HBase
    with pool.connection() as connection:
        table = connection.table(table_name)
    result_from_hbase = table.rows(ids_as_list)
    # Go through all records from the HBase
    for cursor_tuples in result_from_hbase:
        current_element = {}
    [...] # Prepare data structure for HBase
    return result

Listing 3.2: Odoo Model Read Function

3.3.4 Search

When the user navigates through Odoo web pages, Odoo generates search request for any model that has been accessed by the user. These requests are generated by Odoo to determine the ids of the records that should be viewed to the user. After that, Odoo invokes read requests to get the needed information from each model. Likewise, any search that the user may execute on Odoo interface, Odoo fires a search request to specify the matched records, followed by read requests to view the matched records information.

As explained in earlier section, each search request generates a dynamic SQL select statement. As a result of that, we implemented this search function to connect to Phoenix instead of direct communication with HBase.

Phoenix provides a Java library to enable Java clients to communicate with it, but for other programming language Phoenix version 4.4 introduced QueryServer as its REST API. In this thesis, we use Phoenix version 4.7, which is compatible with HBase v1.2 and its QueryServer supports long queries generated by Odoo. To connect to Phoenix QueryServer from Python, we use open source library called PhoenixDB [Lal]. We modified this library to send the data of the HyperText Transfer Protocol (HTTP) requests in the HTTP body instead of the HTTP header. Because, Odoo sends queries with high number of parameters which cannot fit in the HTTP header.

The steps of search request related to ‘mail.message’ models in our approach is demonstrated by the sequence diagram Figure 3.8. First, the user initiates a search request,
for example: Filtering the messages that contain a specific keyword in their title. Then, Odoo front-end forwards the request represented as search domain, mostly using an Asynchronous JavaScript and XML (AJAX) request, to the back-end to be handled by ‘HBaseORM’. ‘HBaseORM’ gets the generated SQL statement and updates it to match Phoenix exact syntax. After that, the query will be sent to Phoenix over the QueryServer. Finally, Phoenix obtains the required information from HBase and returns the results to Odoo.

3.4 System Architecture

In Figure 3.9, we illustrate an overall view of our modified version of Odoo. Odoo executes all requests using normal Odoo ORM layer and PostgreSQL. Only requests for HBase based models (“mail.message”, “mail.notification”, “ir.attachment”) are evaluated by ‘HBaseORM’.

‘HBaseORM’ sends all CRUD operations to the HBase cluster directly through HBase Thrift API. On the other hand, ‘HBaseORM’ executes search requests using Phoenix layer via the QueryServer. Every information stored in HBase database is managed by the Hadoop distributed file system.
Figure 3.9: System Architecture
4. Evaluation

In this chapter, we represent the evaluation of NoSQL enhanced Odoo against the regular Odoo system. In Section 4.1, we describe the Hadoop cluster that we used to evaluate the system. Furthermore, we list the cluster properties which have important impacts on the performance of this modified version of Odoo. Then, in Section 4.2, we explain why we use Thrift API as a bridge between HBase and Odoo. After that, in Section 4.3, we demonstrate the data that we used in our experiments.

To make sure that our modified Odoo returns correct results as the regular Odoo system we made a unit test evaluation which we explain in Section 4.4. In Section 4.5 and Section 4.6, we show a performance comparison between the two systems on both front-end and database system levels.

4.1 Hadoop Cluster Implementation

The Hadoop cluster that we use to make the evaluation consists of four nodes: One master node and three slave nodes.

As shown in Figure 4.1, we use the master node as HDFS master and HBase master, also this master node contributes as a slave for both HDFS and HBase. The slave nodes are both datanodes and HBase regionservers.

The ThriftServer and the QueryServer daemons run on the master node. In Table 4.1, we list the properties of the cluster’s nodes. Moreover, in Table 4.2, we view the properties of the hardware server which runs PostgreSQL database system.
4. Evaluation

Figure 4.1: Hadoop Cluster

Table 4.1: Cluster Nodes Specification

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>Ubuntu 14.04LTS 64-bit</td>
<td>Intel Core i7-5600 CPU @ 2.60GHz x4</td>
<td>12 GB</td>
</tr>
<tr>
<td>Slave1</td>
<td>Ubuntu 14.04LTS 64-bit</td>
<td>Intel Core i7-5600 CPU @ 2.60GHz x4</td>
<td>12 GB</td>
</tr>
<tr>
<td>Slave2</td>
<td>Ubuntu 14.04LTS 64-bit</td>
<td>Intel Core i7-4510 CPU @ 2.00GHz x4</td>
<td>12 GB</td>
</tr>
<tr>
<td>Slave3</td>
<td>macOS Sierra 64-bit</td>
<td>Intel Core i7 CPU @ 2.20GHz x4</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Table 4.2: PostgreSQL Server Specification

<table>
<thead>
<tr>
<th>Name</th>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>Ubuntu 14.04LTS 64-bit</td>
<td>Intel Core i7-5600 CPU @ 2.60GHz x4</td>
<td>12 GB</td>
</tr>
</tbody>
</table>
4.2 HBase Client API Evaluation

We need to access HBase database system using Python, so we need a client API that represents a bridge between our code and HBase. HBase has several client API [Geo11], such as: REST, Thrift and Avro. The main APIs which are supported by the community are REST and Thrift APIs:

- **REST API**: Establish the communication between HBase and the client via HTTP request and HTTP response. It requires data to be wrapped by JSON or XML format and to be encoded as Base64.

- **Thrift API**: Originally, it is developed by Facebook. Then, it becomes an Apache project, to enable a transparent cross-language development [Apa]. Thrift is developed to enable lightweight, quick communications and provides access to row data.

To decide which one of these two APIs we should use, we made an experiment to compare between the process time required to insert 10 thousands rows of ‘mail_message’ into a standalone HBase version using Thrift API and using REST API. We repeated the experiment five times. Then, we calculated the average. The process time to insert the rows using Thrift API takes around 46 seconds in average. While the needed time to insert these rows via REST API is almost 62 seconds. Since Thrift API is quicker than REST API, we used Thrift in our system.

4.3 Data Characteristics

We use the data of an InitOS GmbH customer to compare the performance of the two systems. We have data for three months of real use of Odoo system. To test the system with larger amount of data, we duplicated the dataset related to messages and attachments several times until we reach five million records in messages table.

In Table 4.3, we provide an overview about the original and the duplicated data. In the following experiments, we start with the original data set and measure the performance. Then, we duplicate the data. We repeat these steps until we reach the target number of records.

<table>
<thead>
<tr>
<th>Name</th>
<th>Original No.</th>
<th>Duplicated No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail Message</td>
<td>17,995</td>
<td>5,002,610</td>
</tr>
<tr>
<td>IR Attachments</td>
<td>8,153</td>
<td>2,266,534</td>
</tr>
<tr>
<td>Active User</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

After loading the data into HBase, HBase stores the data in HFiles and Hadoop distributes it equally between the datanodes. In Figure 4.2, we show a snapshot of the
Hadoop control panel, this snapshot illustrates detailed information about the live datanodes of our cluster, such as:

- Nodes name.
- The capacity and the used storage space of the datanode.
- The number of blocks which assigned to each datanode.

![Figure 4.2: Hadoop Data Distribution](image)

In the figure, we can notice that we have four datanodes. All nodes are running ('In Service' state). The file blocks are distributed almost equally between the nodes with average 43 blocks per node. However, we can observe that Hadoop does not use the capacity storage fairly among the nodes.

### 4.4 Unit Test Evaluation

Unit testing aims to test if a small, full functional portion of software, called unit, gives the expected results.

We used this test to check if our changes to Odoo return correct results (with respect to BASE properties) as the returned results of regular Odoo. To develop the test we use:

- OERPLib [OER]: Python library that can call Odoo back-end functions, such as: Create, update model instance and search the model using Odoo search domain.
- Unittest2 [Col]: Python package for writing unit testing code.

After loading the data into HBase, we tested each function that we had overridden in mail messages and attachments modules. We executed the function on normal Odoo; after that, we executed the same function with same input data on our modified version of Odoo. Then, we compared the two results. If the results are equal, then the test was success. Otherwise, we fixed the problem and repeated the test again until we got a successful result.
4.4. Unit Test Evaluation

```python
def test_TestOdoo(self):
    oerp = oerplib.OERP(server='***',
                        database='Normal', protocol='xmlrpc', port=8069)
    oerp_hbase = oerplib.OERP(server='***',
                              database='Normal_HBase', protocol='xmlrpc', port=8071)
    [...]
    n_odoo = oerp.read('ir.attachment', [36], ['id', 'datas_fname'])
    h_odoo = oerp_hbase.read('ir.attachment', [36], ['id', 'datas_fname'])
    self.assertEqual(n_odoo[0]['datas_fname'], h_odoo[0]['datas_fname'])
```

Listing 4.1: Search Attachment Unit Test

The Listing 4.1 presents test case for testing the search function in attachments model:

We create two connections to Odoo, one for normal Odoo and the other for Odoo with HBase. Then, we send same request to both systems which is getting the name for the attachment with id ‘36’. Finally, we check if both names are equal using the ‘assertEqual’ function.

In Figure 4.3, we show the result of executing the test in Listing 4.1. The successful test should return the time of executing the test and ‘OK’ to tell us that our code passed the test.

Figure 4.3: Unit Test Successful Result
4.5 Front-end Evaluation

Our focus in the front-end evaluation is the time Odoo takes to fully load a web page. We choose four pages to evaluate the performance of both Odoo systems, these pages are:

- User’s new messages page: It views the new messages which the logged in user has received.
- User’s sent messages page: This page views the messages that the user sent.
- View messages page: In this page, the user can list all messages which stored in the system. The list of the messages is divided into pages and the limit for each page is 80 messages.
- Filtering messages based on title: We use the view all messages page to view the messages that contains a specific word in their title. Also, the messages are viewed 80 messages per page.

We choose the first two pages because they are the most visited pages by a user and they require multiple queries on ‘mail_message’ table. The other two pages are administrative pages that allow the administrator to view all stored messages in the system and provide the ability to filter them.

In all our experiments, we start with the regular database load of our use case; then we multiply the data for attachments, mail messages and all related information until we reach 5 million records in mail message table. Also, we make two measurements in each step for our modified version of Odoo one with HBase cluster of four nodes and the other with three nodes to study the effect of the nodes number on Odoo performance. For each value in our experiments, we repeat the test five times and report the average value.

In Figure 4.4 and Figure 4.5, we outline the time (in seconds) which it takes to load the new messages and the sent messages for a user by each system. The charts illustrate that the performance of Odoo with PostgreSQL is better when the records are less than three million. With less than three million records the communication overhead between the HBase nodes overcome the performance improvement. At four million records, HBase shows a slightly improvement over PostgreSQL, but with five million records the time which normal Odoo takes to view the user’s new messages is double the time our version of Odoo needs with four HBase cluster.

As show in the figures, at five million records, HBase with four nodes cluster is faster that HBase with cluster of three nodes. For instance, loading the new messages using four nodes cluster takes 12.42 seconds while three nodes cluster needs 15.57 seconds. That indicates, with a larger amount of data, the performance of HBase-Phoenix increases when we increase the number of nodes.
4.5. Front-end Evaluation

Figure 4.4: User’s New Messages

Figure 4.5: User’s Sent Messages
On the contrary, when the number of records is less than approximately 2 millions records, three nodes cluster is faster than four nodes cluster. When we debugged this issue, we noticed that Phoenix generates more parallel processes with higher number of nodes and with larger data size. Then, to calculate the final result, Phoenix merges the intermediate results at the node which initiates the request into one final result. So, when increasing the number of the intermediate results, the merging time may increase.

If the enhancement of the actual data processing with more parallel process does not cover the latency of merging the intermediate results, the processing time of the request will increase.

In addition to that, the distribution of the data between nodes may affect the result. Finding the actual cause for the more latency of four nodes cluster needs more experiments. This experiments are outside the scope of this thesis.

To conclude, if the size of data is large enough, increasing the number of nodes in the cluster will increase the performance of the overall system. However, if data size is small, the communication and synchronization between nodes in the cluster will reduce the performance of the system.

In Figure 4.6, we present a performance comparison between regular Odoo and our version of Odoo to view the messages that are stored in the system. In Figure 4.7, we view the result of a comparison between the two systems to filter the messages which contain a specific word in their subjects. We can notice that the performance of regular Odoo is better in both cases for all data volumes, even with large amount of data. We can notice that the gap between the performance of the two systems reduces with a larger amount of data.

The latency in our Odoo implementation is caused by the high number of HTTP requests that are sent from the Odoo front-end to Odoo back-end. These requests are sent to obtain the required information to view the page. Our modifications to Odoo core increase the number of communication messages between the front-end and the back-end.

When defining many to one or many to many relationship between two models, Odoo provides the relationship with a property called ‘auto_join’. If this property is ‘True’, whenever a request made to one model that needs information from the other, Odoo generates an ‘Inner Join’ SQL statement between the two models’ database tables to get the required information. Otherwise, if ‘auto_join’ is ‘False’, Odoo sends several SQL statements for each table independently to get the same results as if the ‘Inner Join’ query is executed.

To remove the dependencies between our three modified models ‘mail_message’, ‘mail_notification’, ‘ir_attachments’ and other Odoo models, we set all ‘auto_join’ properties link to foreign keys in one of the modified models to ‘False’, such as: ‘mail_mail’, ‘mail_thread’ and ‘res_user’. These modifications force Odoo to send many requests to load the view messages page or to filter these messages.
4.5. Front-end Evaluation

Figure 4.6: Processing Time for View Messages Page

Figure 4.7: Processing Time for Filtering the Messages on Title
In Figure 4.8 and Figure 4.9, we present the difference between the HTTP requests that are sent to view messages by our modified Odoo and by regular Odoo respectively. These figures are taken from Firebug plugin [HOrG].

From the figures, we can notice that the search request is executed faster in the modified Odoo. As shown in Figure 4.8, after Odoo front-end executes the search request, it sends several requests to the back-end to retrieve the missing information. In contrast, we can notice in Figure 4.9 that, after the search request, Odoo front-end starts to retrieve the information for other parts of the web page. In modified Odoo, the processing time for the additional requests sums up to increase the time needed to load the page more than the time needed in regular Odoo.

![Figure 4.8: Modified Odoo Front-End Requests](image)

![Figure 4.9: Regular Odoo Front-End Requests](image)

In addition to previously studied pages, the web pages that present detailed information about a model which is integrated with ‘mail_message’ or ‘ir_attachments’ are also important in our experiment. The time to view such pages is almost the same in both systems. As an example of these pages, the view product page that is shown earlier in Figure 2.7 in Section 2.2.3.

After analyzing the view product page, the query that Odoo performs related to ‘mail_message’ model is:

```sql
SELECT fields
FROM mail_message
WHERE RES_ID = id AND MODEL = 'Model_Name';
```

With five million records in ‘mail_message’ table, in average, PostgreSQL executes this query in $0.035$ seconds while Phoenix needs $0.089$ seconds.

In case any message has attachments, or if the product has attachments, Odoo sends the same query on ‘ir_attachment’ table to the data storage system which has approximately the same latency as in ‘mail_message’.
4.6 Database Systems Evaluation

In this section, we evaluate the performance of PostgreSQL and Hadoop ecosystem to count the number of the messages that has the word 'frage':

```sql
SELECT COUNT(1)
FROM mail_message
WHERE BODY ILIKE '%frage%';
```

Our use case database is in German language, we selected the word 'frage' because it is a non-stop word and it is one of the words that are used a lot in our test data set.

The database systems need to perform this query, a full-text search in the body of each message and full table scan to go through all stored messages. This query requires a powerful processing capability with high volumes of data.

In Figure 4.10, we illustrate the response time needed to perform the query by PostgreSQL and HBase-Phoenix with four nodes cluster. It is noticeable that PostgreSQL with a smaller amount of data is better than HBase-Phoenix. At almost two million records of mail messages, Phoenix starts to perform better than PostgreSQL. The performance of PostgreSQL is reduced greatly with larger amount of data, the response time increase between four and five million records is almost 45%, while Phoenix shows better adaptation with increasing the data size.

![Figure 4.10: Filter Messages on Body Content](image-url)
Phoenix divides the table into several chunks, then, it creates several processes in each node in the cluster to process the chunks in parallel. Finally, each process returns its sub-result to the client to merge it into the final result. This steps improves the performance to answer a specific query, but if the sub-results are large that may produce an overhead in the client.

The Figure 4.11 represents Phoenix execution plan to perform our query with five million records\(^1\). The execution plan indicates that Phoenix handles the request by dividing the table into 19 chunk to perform a full table scan and answer the SQL statement.

![Figure 4.11: Phoenix Execution Plan](image)

To study the effects of handling many requests on the database system performance, we repeat the previous processing time measurements while performing random read on “mail_message” and “ir_attachment” tables.

In Figure 4.12, we present how much time PostgreSQL and Phoenix need to perform the above mentioned query while we are generating simulated read workload.

We developed a Python program that sends search by ID requests to Odoo systems. These search requests target both mail messages and attachments modules. The IDs are generated randomly based on the ID range of the stored data in the systems. We were executing four instances of this program during the running time of the query.

For small amount of data, PostgreSQL is still better than Phoenix with a cluster of four nodes. At two million records, Phoenix starts to perform better than PostgreSQL. Phoenix response time increases slightly with larger amount of data while PostgreSQL response time increases significantly. When the data is large enough, for example five million records, PostgreSQL performance with read workload decreases by 60% less than its performance without any workload. Phoenix shows more scalability than PostgreSQL and to increase Phoenix scalability we can add more nodes to the cluster. On the other hand, to increase PostgreSQL performance and scalability, we need more powerful hardware.

\(^1\)`sqlcline.py` SQL command line provided by Apache Phoenix
Finally, in Figure 4.6, Figure 4.10 and Figure 4.12, we can mark that when the data is almost one million records or less HBase-Phoenix is slower than larger amount of data. We noticed that, if the size of data is small, Phoenix generates less number of the parallel processes. Since the data is distributed among the nodes, a process that runs on a specific node may require data from other nodes to complete its task. Thus, this latency may be caused by the need to transfer data from one node to another. To confirm the reason of this latency, we need to do further analysis and experiments which are not in the scope of this thesis.
5. Conclusion

In this thesis, we suggested a solution to improve a latency in Odoo ERP system. This latency, as InitOS GmbH reported, is caused by two Odoo modules (mail messages, attachments). These two modules can be linked and used by any other Odoo modules. As a result, the sizes of the relational database tables which are related to mail messages and attachments become large during Odoo life-time. This growing size reduces the performance of the Odoo system.

Relational database system was invented by Edgar F. Codd in 1970 [Cod70]. Since then, RDBMSs have been used as primary data storage systems for the developed applications. Web 2.0 applications revealed the needs for more scalable, powerful, distributed data processing systems which led to developing NoSQL systems.

NoSQL systems are promoted as more performed systems than SQL data storage systems. Therefore, we chose to solve the problem of Odoo latency using NoSQL system. We selected HBase-Hadoop as our new NoSQL data storage. To evaluate our new system, we compared its performance with the performance of the regular Odoo system.

To allow Odoo to communicate with Hadoop ecosystem as NoSQL data storage, we developed an Odoo module and change the data model for our targeted Odoo modules (“mail.message”, “ir.attachments”, “mail.notifications”) to be suitable for HBase database system. Odoo is developed to generate dynamic SQL statement, compiling these statements into HBase equivalent queries requires a lot of time. Besides, some SQL operations do not have direct equivalents in HBase query language, such as “Order By”. Hadoop ecosystem provides alternative solution by using one of the ecosystem’s applications that acts as SQL layer over HBase, so we chose Phoenix.

The results of our experiment revealed that our use case application can still use PostgreSQL for long time as better choice than our proposed Hadoop stack. To get the benefits of using our developed solution, large data volumes are required. Otherwise, the time needed to start the query processing and the communication between nodes overhead will make Hadoop ecosystem slower.
Although for the near future PostgreSQL has shown better performance, Hadoop ecosystem has more ability to handle huge volumes of data, increasing number of requests and executing intensive data processing reports.

Our recommendation would be to use Hadoop ecosystem as secondary data storage and store up-to-date data. Thus, Hadoop ecosystem can be used for generating reports that may affect the performance of PostgreSQL. In addition, whenever PostgreSQL performance is reduced the customer can switch to use Hadoop ecosystem smoothly.
6. Future Work

The experiment results are not as promising as we expected. However, integrating an application with big data technologies opens new opportunities by providing powerful data processing framework and increasing the system availability. Also, while analyzing Odoo ORM, we noticed a potential improvement in the way Odoo generates database queries.

Odoo Optimization

To render a requested page from Odoo application, Odoo front-end sends several requests to the back-end to retrieve the information. Some of these requests are limited to a specific piece of information, as a result, the front-end needs many requests to present the page. Nevertheless, these multi-requests are linked to each other’s and can be grouped in one request, which may have good impact on Odoo performance.

Further Evaluation

The performance of NoSQL systems can be improved by adding more nodes to the cluster. In the future, we can add more nodes to our suggested Hadoop stack and re-evaluate the system performance with different data sizes. More deep analysis may be required, to study the effects of data distribution and replication on HBase-Phoenix performance.

Additionally, there are other Hadoop distributions that have different Hadoop stack which can be used in our use case. For example, Cloudera provides a Hadoop distribution that contains Impala\(^1\) as an SQL layer over HBase. We can use this distribution in our module. Then, we compare the performance between the different Hadoop stacks.

\(^1\)https://www.cloudera.com/documentation/enterprise/5-8-x/topics/impala_hbase.html
**Hadoop Ecosystem Potential Uses**

Storing the data in HBase allows us to develop complex data processing tasks using the distributed programming model MapReduce. For example:

- Performing data analysis on un-structured data, such as attachments.
- Creating integrated reports that contain un-structured and structured data, such as: Analyzing messages information, messages text and linked attachments.

Further, we can enhance our Hadoop stack full-text search functionality by integrating HBase-Phoenix with *Elastic Search*\(^2\) [Fou14]. Elastic Search is an open source, distributed search engine.

On the other hand, web logs contain huge amount of important information that can be used to provide better services. Web logs are un-structured data which is difficult to be processed. We can implement MapReduce jobs to analyze the web logs. Then, the results can be stored in HBase and use ‘HBaseORM’ module to access these results via Odoo interface.

\(^2\)https://www.elastic.co/
A. Appendix

Hadoop and HBase Configuration

```
<configuration>

<property>
    <name>fs.default.name</name><value>hdfs://master:9000</value>
</property>

<property>
    <name>hadoop.tmp.dir</name><value>PATH-TO-LOCAL-FILE</value>
</property>

<property>
    <name>ipc.server.tcpnodelay</name><value>true</value>
</property>

<property>
    <name>ipc.client.tcpnodelay</name><value>true</value>
</property>

</configuration>
```

Listing A.1: Hadoop core-site.xml
Listing A.2: Hadoop hdfs-site.xml
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>HDFS-STORAGE-PATH</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.master</name>
    <value>master</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.maxClientCnxns</name>
    <value>1000</value>
  </property>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>master,slave,shadi</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.clientPort</name>
    <value>2181</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>PATH-TO-LOCAL-FILE-STORAGE</value>
  </property>
  <property>
    <name>phoenix.query.timeoutMs</name>
    <value>600000</value>
  </property>
  <property>
    <name>hbase.regionserver.wal.codec</name>
    <value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec</value>
  </property>
  <property>
    <name>phoenix.query.targetConcurrency</name>
    <value>200</value>
  </property>
  <property>
    <name>phoenix.query.maxConcurrency</name>
    <value>300</value>
  </property>
  <property>
    <name>hbase.ipc.server.tcpnodelay</name>
    <value>true</value>
  </property>
  <property>
    <name>phoenix.connection.consistency</name>
    <value>timeline</value>
  </property>
</configuration>

Listing A.3: HBase Configuration hbase-site.xml
Bibliography


Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den