UiA University of Agder Norway

A comparison of database systems for rapid search results in large amounts of leaked data

Nicolai Prebensen

SUPERVISOR

Morten Goodwin

Master's thesis
University of Agder, 2019
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

A large portion of the information on the Internet is stored in databases, since databases are the natural place among online services for storage of user information. Hackers occasionally breach web sites and publish database records online, resulting in personal user information leaked in the public domain and available for abuse by malicious actors.

Several web services offer the opportunity to search in numerous data breaches with as much as six billion records or more and response times less than one second for each single query.

This thesis investigates performance optimization methods that can be applied to the database systems MySQL Percona, MongoDB Percona, Splunk and Elasticsearch to provide rapid search results on leaked data with average hardware specifications.

Background information on database systems and core functionality is described along with possible configuration options for operating system tuning. State-of-the-art database solutions for handling big data are also presented.

The database systems have been extensively tested and show that all database systems can be tuned for improved performance. MySQL and MongoDB delivers query results in almost real-time for exact searches when using indexes, while Splunk is among the faster solutions for both exact and wildcard queries.

Research confirms that Elasticsearch is the fastest performing solution for searching leaked data with an average response time 1.58 seconds on data sets containing between 10 and 100 million records.

Acknowledgement

This thesis concludes my final work as a master-student of information- and communication technologies at the University of Agder, Grimstad.

Primarily, I would like to thank my supervisor **Dr. Morten Goodwin** for the guidance during the past months of writing this thesis. Your guidance has been very helpful, and your constructive feedback has helped me greatly in the process of completing my thesis.

When my motivation has not been on top, family, friends, and my girlfriend have supported me and pushed me to finish my thesis, which I highly appreciate.

Additionally, I would like to thank especially two lecturers at the University of Agder.

Thank you to **Sigurd Kristian Brinch** for allowing me to host my server at the University and providing me VPN access for managing my server remotely. This has been of great help for the implementation and testing of my experiments.

Thank you to **Arne Wiklund**, a lecturer at the University of Agder with broad experience with database systems, for helping me with difficult problems and guidance during the course of writing this thesis. Especially with Apache Hadoop and technical discussions.

My colleagues at NTT Security Arendal has also been helpful with constructive feedback and support.

Lastly, I hope this thesis will offer some new knowledge regarding database performance selection and optimization techniques for you as a reader.

Nicolai Prebensen
Grimstad, May 24th 2019

Table of Contents

List of Tables

List of Figures

Listings

Part I

Research Overview

Chapter 1

Introduction

1.1 Motivation

The amount of data on the internet increases daily as new users sign up and register for services online. Typically, when registering for an online service a user is required to provide data that can be used as login information such as a username, password, and e-mail address at a minimum. Other sites may also require users to provide full name, address and phone number for verification purposes, all of which get stored in large databases. From time to time web sites get breached and databases are publicly leaked on the internet, resulting in personal information potentially ending up in the hands of malicious actors. There is several where such information is leaked. This thesis examines how to set up databases to efficiently search leaked data.

Depending on which web site has been hacked and had its database leaked, the consequences can be critical as malicious hackers are interested in obtaining this information for use in attacks or reconnaissance of targets. However, not only malicious actors are interested in obtaining the leaked databases. A number of white hat hackers and security researchers use the same information (leaked data) for their services. These services offer searches in breached data for end users to determine if their personal information and passwords have been leaked to the public.

The web site *haveibeenpwned* by Troy Hunt is one of these services and contains approximately 6.9 billion breached records at the time of writing [1]. By using *haveibeenpwned* users may enter their e-mail address to check whether they are part of a breach, and see which breaches contain their e-mail address. Other sites offer similar services but with less respect to personal privacy such as *WeLeakInfo* [2] or *LeakedSource* [3], where users are presented clear-text usernames, passwords, IP-addresses, phone numbers and other available informa-

tion for their search query.

To query such large amounts of data and returning results within seconds, a stable database system and optimization methods are required.

The motivation behind this thesis is to investigate and optimize database systems to return rapid search results on very large sets of data. In this thesis, the focus is on breached data and achieving rapid results on search queries. The resulting approach in this thesis can however be adjusted to achieve similar results on other types of data.

1.2 Thesis Definition

The primary motivation for this thesis is to evaluate the performance of four database systems and determine the best possible solution for achieving the fastest possible search results on large sets of data. The database systems are initially evaluated on a default installation, followed by another evaluation after optimization techniques are applied.

1.2.1 Thesis Goals

Goal 1: *Determine which database system should be used to achieve the fastest possible search results on large amounts of leaked data.*

Goal 2: *Utilize optimization methods to achieve rapid search results when using inexpensive hardware that is typical for the average user.*

Goal 3: *Achieve rapid search results without the use of indexes in MongoDB and MySQL for exact- and wildcard searches.*

Goal 4: *Achieve a query response time of fewer than five seconds for the maximum records tested.*

1.2.2 Hypotheses

Hypothesis 1: *MongoDB Percona can be optimized to provide better performance than Elasticsearch, MySQL Percona or Splunk.*

Hypothesis 2: *Proper optimization can result in a significant performance improvement.*

1.2.3 Summary

The goal of this thesis is to find a solution for achieving rapid search results on large amounts of leaked data for both exact and wildcard queries. Another goal is to provide setup instructions and optimization techniques that can be applied to achieve the fastest possible search results, without the use of expensive hardware.

1.3 Scope & Limitations

The selection of database systems is based on certain limitations for the scope of this thesis. Primarily only open-source solutions are tested except for one, which offers a free 30-day enterprise trial*. Each database system is evaluated in terms of performance and optimization methods applied respectfully.

The following systems are evaluated:

- MySQL Percona
- MongoDB Percona
- Elasticsearch
- Splunk*

It must be emphasized that each database system is focused on as a whole. Low-level storage algorithms and source code of the database systems are not analyzed in depth, but rather the available options for tuning performance.

All database systems are tested with an equal amount of data and several optimization options to investigate the differences in performance respectfully. Initially, the scope of maximum records was set to six billion but was adjusted to 100 million records during experimental testing.

All database systems are installed and evaluated on a single machine to represent a computer setup that is similar to that of an average user. Clustering is therefore not included in the experiments.

1.4 Research Methodology

There exist several methodologies that can be followed when writing a research paper. *Scientific Methods in Computer Science* is a paper written by Crnkovic [4] that identifies research methods in computer science with relations to development and technology. While the ideal of science is physics, the same ideal does not apply to computer science. Therefore, Crnkovic presents three methods that can be utilized for computer scientific research:

The first method is **the theoretical method** which follows a traditional way of logic, typically by using mathematics and various levels of abstractions in order to build logical theories. "One of theoretical Computer Science's most important functions is the distillation of knowledge acquired through conceptualization, modeling, and analysis" [4]. This method can be used when attempting to find solutions to performance issues or the design of algorithms.

Second, **the experimental method** is used when a computer scientist observes, explores and tests theories by using real-world experiments. By observing the result of the experimental testing against a theoretical prediction, the computer scientist can obtain new knowledge. Additionally, when performing experiments the computer scientist will experience the results in person compared to using the theoretical method.

The third method is **the simulation method** which allows complex problems to be simulated and visualized, giving a clear overview of the results. This method can be helpful when experimental testing is not significant or appropriate for the problem to be solved.

The research in this thesis follows the experimental method. However, by these definitions, both the theoretical method and experimental method could be utilized. The reason for selecting the experimental method is due to testing on real-world hardware to experience response times in person.

1.5 Contributions

Primarily this thesis provides an overview of a variety of database systems for handling large amounts of leaked data.

This thesis also provides a generalized solution and setup for database systems that can be used to achieve rapid search results on large amounts of data. The contribution consists of experimental results and detailed instructions on how to set up a database environment using inexpensive hardware. The solution presented will hopefully be of use for others seeking rapid database performance.

Although the solution focused on breached data, similar results should be achievable for other types of data.

By this, we present the first detailed comparison on setups for rapid search results on large sets of leaked data for the database systems in question, including results.

1.6 Thesis Outline

Chapter 2 provides background theory on data structures, database models, storage engines and the selected database systems.

Chapter 3 covers research on state-of-the-art optimization techniques that can be applied to the operating system or specific to each database system. Existing research on the thesis topic is also presented in this chapter.

Chapter 4 outlines the technical implementation and optimization methods used to perform the experiments, ultimately achieving the goals of this thesis.

Chapter 5 presents the results from the experimental testing and proposed optimization techniques for achieving rapid search results on large amounts of leaked data.

Chapter 6 concludes the hypotheses and goals of the thesis.

Chapter 7 outlines further research that can be done in the future.

Chapter 2

Background

During the recent years of computer evolution databases have played a major role for the storage of data. In the early years of computers, punch cards were used for data storage as they offered a fast way of entering and retrieving data [5]. Today multiple data storage solutions exist and database systems are primarily used for this task.

Almost every major web service require a database to store user information. To achieve this task database systems are used, allowing a person or application to organize, retrieve and store data efficiently and conveniently [5].

There exists multiple database models and several types of each database model. Relational databases typically handle structured data while *NoSQL* databases are used for storage of unstructured data, often in document stores. *Search Engine* database systems also exist with the main purpose of performing full-text searches.

Having chosen the correct database system is important due to performance as the amount of data may increase over time. Not all database systems perform equally which potentially can result in slow query response times if the database system is not optimized or used for its intended purpose.

In the following sections, the different types of database models and types are explained. The four database systems used in this thesis are also presented.

"A database is a set of data stored in a computer.
This data is usually structured in a way that makes
the data easily accessible." - *Codecademy* [6].

2.1 Data Structure

Not all data can be stored in a single uniform way due to the vast diversity of data formats. Data may have a certain structure that is recognizable and easily searchable, which is referenced to as *structured data* [7].

On the other hand, there is *unstructured data* which has a structure that does not necessarily fit into rows or columns the same way structured data does. Almost every data set is unique, therefore options must be considered in order to select the most suitable database for the data to be stored.

As described by *DataMation*, structured data consists of easily searchable patterns with defined data types. Unstructured data on the other hand, consists of data such as audio, video and social media postings that are not easily searchable [8].

Some data sets consist of a simple structure that easily can be handled while other data sets can be more complex. Data may therefore be categorized as structured or unstructured.

2.1.1 Structured Data

Structured data is a term used to describe data that generally has a format for big data and a defined length [9]. Typically for structured data is that an identical structure is occurring repeatedly in the data set. Due to this structured data is often stored in rows and columns in a relational database management systems (RDBMS), where queries can be performed by using Structured Query Language (SQL).

Examples of structured data include numbers, dates and strings that are easily distinguishable and may easily be searched for regardless of data source. Other examples of structured data stored in relational databases include sales transactions, ATM activity and airline reservation systems [8].

Structured data can by this be considered data that is organized to a high degree, making it perfectly suitable for relational databases.

2.1.2 Unstructured Data

Unstructured data is data that can not be predicted and is normally not as easily searchable in comparison with structured data, as it may differ greatly with every occurrence and does not follow a uniform format. Examples of unstructured data are e-mails, videos, images or

blog posts [8]. Due to the differences that may occur this type of data is often stored in NoSQL databases, and very often in document stores [10].

Unstructured data do however have an internal structure but is not structured by predefined schemas or models, making it suitable for storage within non-relational databases such as NoSQL systems [8].

2.1.3 Semi-structured data

Semi-structured data is a combination of both structured and unstructured data. This type of data can have a self-describing structure such as field names for the same type of data but does not necessarily follow a certain structure. Every record can contain attributes that vary and is unordered within a class. As semi-structured data has a certain level of organization of properties, it can be stored in relational databases if the data is analyzed [11].

2.1.4 Structure Comparison

Structured data is a term for data containing a high level of organization, which in turn allows the data to easily be stored in a relational database and queried quickly. Unstructured data is essentially the opposite [8] and semi-structured data is a combination of the two.

Another difference between the data types is the analysis of the data, where multiple tools exist for structured data analysis, but few tools exist for unstructured data. For searches, there is an advantage with structured data, as only a specific field has to be searched through, compared to a time consuming sub-string search through unstructured data, or having to analyze semi-structured data.

Unstructured data accounts for approximately 80% of enterprise data and has an increasing annual growth rate. Examples and differences in structured and unstructured data can be seen in Figure 2.1.

	Structured Data	Unstructured Data
Characteristics	• Pre-defined data models • Usually text only • Easy to search	• No pre-defined data model • May be text, images, sound, video or other formats • Difficult to search
Resides in	• Relational databases • Data warehouses	• Applications • NoSQL databases • Data warehouses • Data lakes
Generated by	Humans or machines	Humans or machines
Typical applications	• Airline reservation systems • Inventory control • CRM systems • ERP systems	• Word processing • Presentation software • Email clients • Tools for viewing or editing media
Examples	• Dates • Phone numbers • Social security numbers • Credit card numbers • Customer names • Addresses • Product names and numbers • Transaction information	• Text files • Reports • Email messages • Audio files • Video files • Images • Surveillance imagery

Figure 2.1: Examples and differences in structured and unstructured data [8].

The data experimented with in this thesis is considered structured as the same fields repeatedly occur in the generated data set. As publicly leaked data is considered semi-structured because it originates from multiple databases using different data structures, the data must normally either be pre-processed or inserted in a database without a uniform structure.

Pre-processing real leaked data will provide the same results as automatically generating data for the experiments in this thesis. An example of the generated data can be seen in Figure 2.2.


```
"_source": {
    "password": "gtNeKgkXdRwDdQ63Y1HZ",
    "leak": "master_leak_1m",
    "salt": "9ff89",
    "email": "jEfrxHus@hotmail.co.uk",
    "password_hash": "9e54945bbec0860ac9586f7675c26da1
    "username": "jEfrxHus",
    "ip": "199.69.74.202"
}
```


Figure 2.2: Example of generated leaked data from the datagenerator script.

Although this thesis investigates data that can be of various structures there is no correct solution for the selection of database system solely based on this, as the selection of database system primarily depends on the purpose of the data.

2.2 Database Models

Selecting a database to use depends on the data to be stored as mentioned in the previous section. This section presents several database types for both the relational- and NoSQL database models.

> "One of the most fundamental choices to make when developing an application is whether to use a SQL or NoSQL database to store the data." - *Serdar Yegulalp, InfoWorld* [12].

2.2.1 Relational Databases

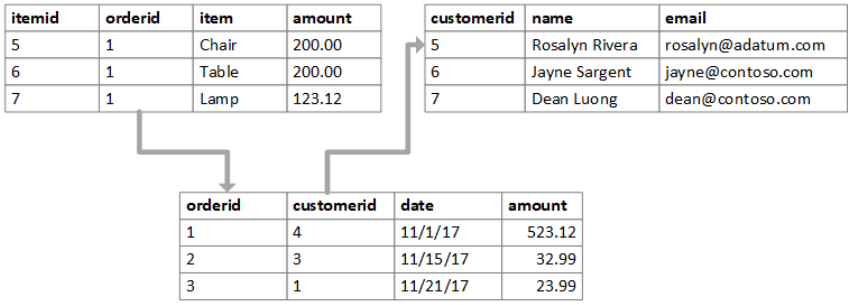
The most frequently used database type is the relational database model and is based on a structure that allows users to access and identify data in relation to other data in the database [6].

The structure of a relational database consists of tables of rows and columns, as well as both primary- and foreign keys used to reference data in separate tables. These *rules* form the database schema which contains information about the database structure. The database type follows a relational model, hence the name "relational database". A table can contain multiple columns, where each column is labeled and defined to hold a certain data type such as integers, strings, timestamps or other data formats.

An entry or record in a relational database is stored in a single row consisting of one or more attributes, based on the defined columns. As the data can be stored in different tables depending on its purpose, redundancy can be defeated as identical information does not have to be stored in multiple places. An example of this is tables containing home addresses, where zip-codes can be stored in a separate table and be referenced using a relation.

Due to the organized structure of the relational model, data can be reassembled or accessed in multiple different ways without having to reorganize data within the tables [13].

The standard access mechanism for relational databases is SQL (Structured Query Language) which is the most commonly used language for performing database queries and transactions. The syntax in SQL is relatively simple as it is quite similar to the English language. An example of a relational database structure can be seen in Figure 2.3.



itemid	orderid	item	amount
5	1	Chair	200.00
6	1	Table	200.00
7	1	Lamp	123.12

customerid	name	email
5	Rosalyn Rivera	rosalyn@adatum.com
6	Jayne Sargent	jayne@contoso.com
7	Dean Luong	dean@contoso.com

orderid	customerid	date	amount
1	4	11/1/17	523.12
2	3	11/15/17	32.99
3	1	11/21/17	23.99

Figure 2.3: Examples of a relational database structure by Microsoft [14].

Relational databases were for a long time the optimal solution for data storage and maintenance. Due to the reliability in terms of transactions and ad-hoc queries of the ACID (Atomic, Consistent, Isolated, Durable) principle, all transactions were checked before being committed to prevent corruption of the database.

As the data sets of enterprises increased over time some of the reliability features resulted in restrictions, which is the problem that NoSQL attempts to solve by utilizing the BASE (Basically Available, Soft State, Eventually Consistent) principle [15].

2.2.2 NoSQL Databases

NoSQL, also known as *Not only SQL* refers to a database system that does not use the structured query language [16]. Compared to relational databases NoSQL databases are not limited by the same restrictions and structure. There are no schemas, the consistency models are not as strict, and NoSQL data essentially not being relational theoretically allows for faster speeds and performance compared to relational databases.

As suggested by the naming of this database type, traditional databases were not suitable for solving all problems, especially handling large amounts of data.

NoSQL databases were initially introduced when the large tech giants Facebook, Amazon, Google, and Yahoo required new solutions for storing and processing of data from their web services. The NoSQL database technology was introduced as the solution to this problem and can be scaled horizontally across hundreds or thousands of servers [12].

Although NoSQL databases has certain advantages over relational database systems a developer or system administrator must still consider the use-case when selecting a database type. For NoSQL, some of the benefits include caching solutions, faster access to bigger sets of data and less rigid consistency requirements. [16]

The types of NoSQL databases are explained in the following sub-sections.

Key-value Store

Key-value stores are among the most simple database system types. In a key-value store data is stored as tuples consisting of a key and a value. Due to the simple implementation this database type is not frequently used for complex problems, but rather used in certain cases where efficiency is important, exactly due to its simplicity.

One of the cons with key-value stores is that it is not possible to retrieve data if the key is unknown since values cannot be searched for.

Document Store

A document store is a collection containing multiple documents consisting of data represented by fields and does not follow a schema for structure. The documents do not have a uniform structure, meaning documents can have different data fields, unlike relational databases where all records must have the same columns.

Instead of columns only holding single values, columns can contain arrays of data resulting in a nested structure that can be useful for certain applications. Normally, the JSON format is used as a notation in document stores allowing direct processing in applications [17].

Wide Column Store

Wide Column Store is a structure designed to hold large numbers of dynamic columns and is similar to key-value stores except that they are two-dimensional. Column names and record keys are not of a fixed size, which allows the storage of billions of columns for a single record [18].

The implementation of column stores is similar to document stores as no schema is followed, except that column stores follow a fixed format.

Wide column stores "uses tables, rows, and columns, but unlike a relational database systems, the name and format of the columns can vary from row to row in the same table" [19].

Graph Store

Graph stores are different from the other database types. In graph stores data is stored in a graph consisting of nodes and edges, forming a relationship between multiple nodes. Utilizing this *tree* structure makes processing and calculation of data easy and efficient when answering a query.

Search Engines

A search engine is a database system that is purposely created for searching for specific data content.
In addition to being a NoSQL database system, a search engine utilizes certain techniques such as inverted indexes for sorting data in stored documents. Other typical features of search engines include support for complex queries, full-text searches, distributed searches for scalability, and ranking of results [20].

2.3 Storage Structures

A database system consists of multiple components; an API layer for managing the database, a server, and a storage engine. Each database type has a storage engine on a lower level that is responsible for data to be stored. The layers can be seen in Figure 2.4.

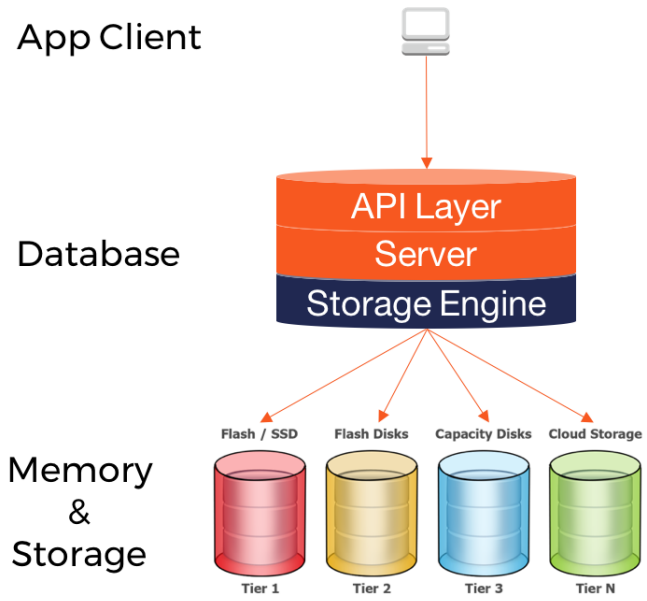


Figure 2.4: Structure of a database system [21].

> "A database storage engine is an internal software component that a database server uses to store, read, update and delete data in the underlying memory and storage systems." [21]

2.3.1 Binary Tree (B-Tree) Based Structure

The binary tree based engine was first introduced in 1971 and is based on a structure that holds information about stored data similar to a tree. Due to the tree structure, the B-Tree engines allow data nodes to quickly be sorted, offering fast insertions, deletions, and searches in logarithmic time [21].

2.3.2 Log Structured Merge (LSM) Tree Based Structure

Over time data sets grew larger resulting in that the B-Tree engine no longer was sufficient due to poor write performance. To overcome this problem database administrators turned to the LSM based storage engine introduced in 1996. The LSM tree data structure is the best fit for large amounts of write operations over an extended period of time [21]. It functions by using "an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort" [22].

Although LSM is considered state-of-the-art today, the B-Tree engine can still compete in terms of read performance.

An overview of which engine structure is used in various database systems can be seen in Figure 2.5.

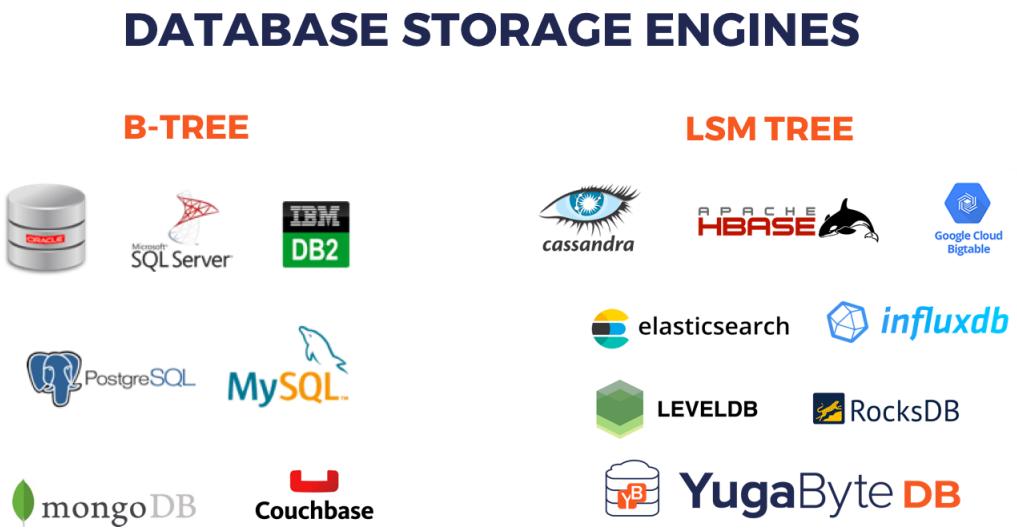


Figure 2.5: Database systems using B-Tree and LSM engines [21].

2.4 Inverted Index

When dealing with large amounts of documents and data in regards to full text searches, an inverted index is often used for analysis and sorting.

An inverted index stores a unique id of each document. When creating indexes, each newly discovered document is given an integer value. A list of terms in the document is also used in the index creation, essentially forming a tuple of document id and terms. This allows the inverted index to return only a set of documents when matching words by using proximity queries. Proximity queries allow matching within k-words, within a sentence, or within a paragraph.

An example of an inverted index can be seen in Figure 2.6.

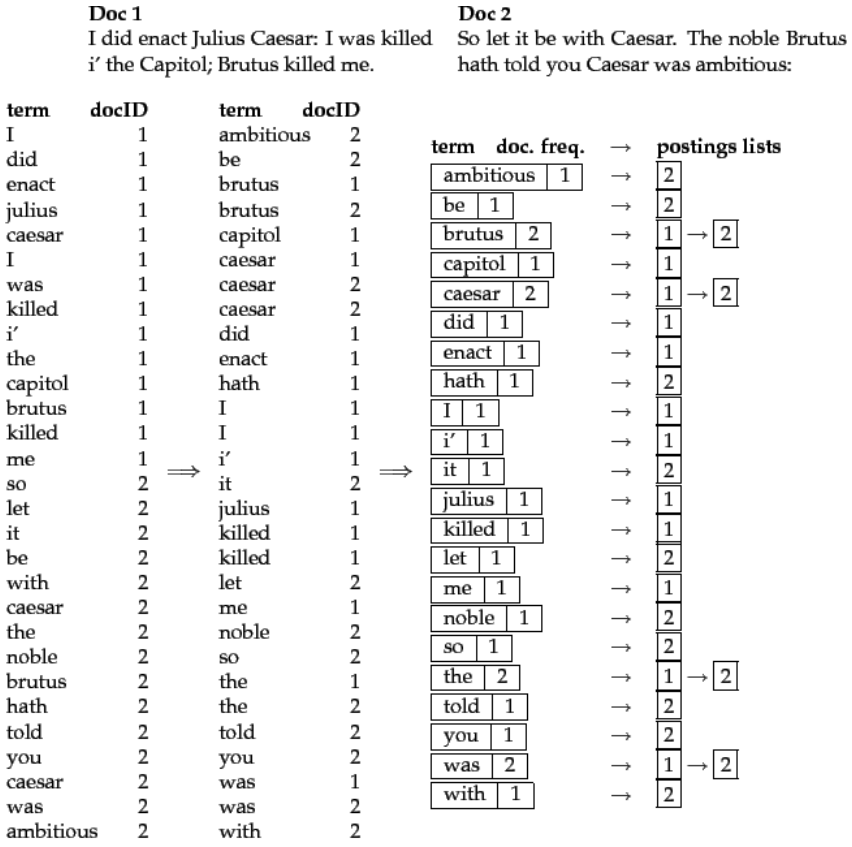


Figure 2.6: Example of an inverted index holding a unique document id and frequency of terms in postings lists [23].

2.5 Database Systems

Database systems, often called database management systems, are software systems that allow programmers and users a systematic way to update, retrieve, create and manage data [24].

2.5.1 MySQL Percona

MySQL is an open-source database administration system using the structured query language and is among the most used relational database management systems. Percona is a company that has developed its own fully compatible fork of MySQL specifically for improved performance.

According to Percona, thousands of enterprises utilize Perconas technology for the gain in performance, availability, concurrency and scalability for their demanding workloads [25].

The owner of leaked data service *Snusbase* [26] states that they utilize MySQL Percona for searches in leaked data.

2.5.2 MongoDB Percona

MongoDB is a widely used NoSQL database with cross-platform functionality, using a document store for data storage. MongoDB is originally developed by MongoDB Inc. but as the database system is open-source, other companies have created forks of MongoDB.

Similarly to the Percona server for MySQL, Percona has developed an optimized server for MongoDB focusing on extremely high performance and reliability [27]. As performance is key in this research, the Percona server for MongoDB is used.

2.5.3 Elasticsearch

The distributed search engine Elasticsearch is growing in popularity due to its capability of solving problems with large sets of data.

Elasticsearch is part of a set of tools, referred to as the *Elastic Stack* with the main purpose of storing and handling data. The search engine utilizes inverted indexes, which results in very rapid full-text searches.

As businesses are interested in retrieving data quickly, Elasticsearch is often a popular NoSQL database solution used for achieving this purpose [28].

The leaked data service *WeLeakInfo* [2] is using Elasticsearch according to their developer. The system has therefore been included in this research.

2.5.4 Splunk

Splunk is an advanced data storage, monitoring, and analysis tool with the capability of collecting, saving and indexing large amounts of data in real-time. This allows users to manage data in a structured way for purposes such as surveillance of log-data, in addition to retrieving valuable information by using dashboards and other visual functionality available in Splunk's web interface.

The storage in Splunk is not a relational database for records and indexes, but instead uses a flat file-based storage method for indexing.

Part II

Analysis & Implementation

Chapter 3

State-of-the-art

Utilizing state-of-the-art performance techniques is an important factor in this thesis. This chapter introduces various performance optimization techniques that can be applied in the host operating system or directly to the configuration files of the database systems.

Existing research articles related to the thesis topic is included in a separate section, presenting comparisons and evaluations of several database systems.

3.1 Performance Evaluation

According to the report *Database Systems Performance Evaluation Techniques* database performance can be evaluated in several ways depending on the desired results. The paper also describes various evaluation methods in detail [29]. For evaluation, Benchmarking tools provide the most accurate evaluation and overhead information.

In this thesis, the goal is to achieve the fastest possible response time on queries as a user. Therefore the database systems have been evaluated by performing queries manually in order to experience the actual response time in person.

3.2 Optimization Techniques

Normally each database system depends on one or more configuration files where settings can be adjusted in attempts to achieve better performance. By default, the configuration files contain settings that are dependent on the hardware specifications of the host machine. Tuning these settings and variables correctly can result in better performance.

General optimization methods also exist which are not specific to the database systems being investigated. The various performance improving methods has been listed in subsections below respectfully.

Although optimization methods related to multiple nodes are not applied in our experimental testing, valuable information on these methods is still included in this chapter for reference.

3.2.1 Scaling

Scaling is a solution that can be used to distribute work over more hardware to maximize performance and computation. This can be done in two ways, either by scaling horizontally, or vertically. The machines that are grouped forms a *cluster*.

Horizontal

Horizontal scaling means that more machines are added into the pool of available resources. An example of this can be multiple interconnected servers deployed next to one another, allowing a single job to be processed with the power of all nodes.

Vertical

Vertical scaling means that more power is added to the available pool of resources, within the same node. This can for instance be more disk space, more computational power (CPU) or more memory (RAM).

For the experimental tests performed in this thesis, scaling is not included as one of the goals of the thesis is to maximize performance on a single node.

3.2.2 Clustering

When a single machine no longer is sufficient for the problem it is designed to solve, more computational power is needed and new nodes are added to the group of available nodes. Briefly explained clustering is the process of scaling a system horizontally by adding more machines to the pool of resources, essentially forming a cluster.

Hot-Warm-Cold-Frozen Architecture

Within a clustered environment storage techniques can be applied for caching data that is frequently asked for, where data is stored in *buckets*. Additionally, the nodes in a cluster may have different hardware specifications. Some nodes can use hard disk drives (HDD) while other nodes can use faster drives such as solid state drives (SSD).

Using the hot-arm architecture, the primary master nodes run on solid-state-drives and are considered *hot* nodes. When the data processed on these nodes has been indexed or reaches a certain age, the data is moved down the node hierarchy to *warm* nodes that can be running on slower drives such as hard disk drives. By using this architecture the best performing nodes will always be used for the most complex tasks to ensure optimal performance. Data that is frequently used will also be kept on the hot or warm nodes, and moved to *cold* nodes when the data bucket reaches a certain age. [30]

3.2.3 Sharding

Sharding is a form of horizontal scaling. By using sharding a database can store data on multiple nodes as a method for handling data growth. When the amount of data reaches the storage limit and a single node no longer can handle the read and write operations required with sufficient throughput, other machines (shards) are utilized [31]. Sharding can also be utilized on a single node.

By using sharding it is possible to create replications, or replica sets. A replica set is a mirror of a shard and consists of a master node and one or more slave nodes that can help handle read operations. Due to this, read-performance can be increased by the use of replica sets. The slaves can however not support in write operations [32].

Sharding is often used as vertical scaling can be expensive. Adding more shards or nodes to a cluster often provides better performance than vertical scaling.

An example of a sharded setup with a master(primary) and slave(secondary) nodes can be seen in Figure 3.1.

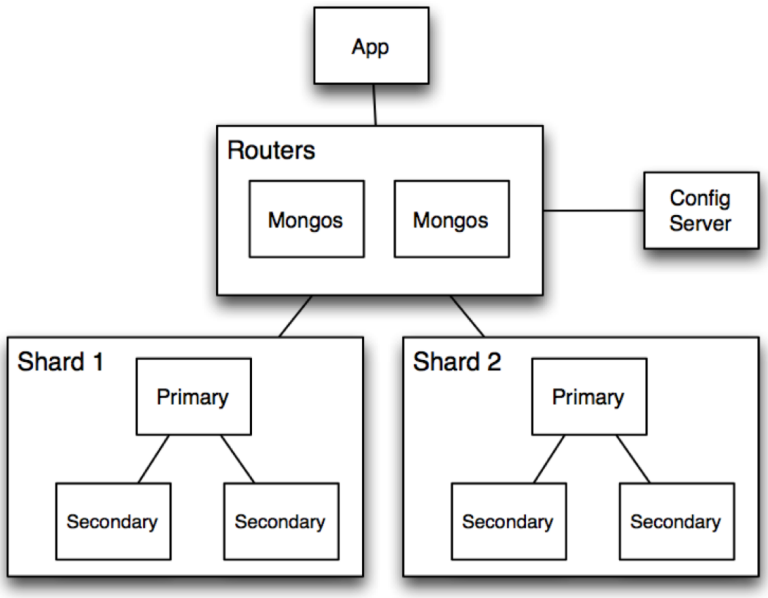


Figure 3.1: Example of a sharded setup with MongoDB [33].

3.2.4 In-memory Storage

In-memory storage is a technique that allows the storage of data directly in virtual memory and functions very well when the full data set fits within the size of available memory.

This is the optimal solution for an increase in performance but requires large amounts of memory. For big data solutions, this optimization technique is not viable.

It is possible to partly apply this optimization method by storing indexes in memory and data on solid state drives for increased performance on larger data sets.

3.2.5 Operating System Tuning

Operating systems handle memory, input/output file processing and, file system operations differently. By adjusting the limitations and settings used by the operating system, faster performance can be achieved for running services. The available options for operating system tuning are listed below.

Disable SWAP Memory

Swapping is a technique used in operating systems when the machine runs out of physical memory. Swapping allows part of the disk in a machine to be allocated and used as memory in such situations.

The memory (RAM) in a machine is significantly faster than a disk in terms of read and write operations. If the maximum memory limit is reached and swapping is utilized, performance will decrease greatly.

Disable Transparent HugePages in Linux

Operating systems use *pages* for applications and services that run in virtual memory and is mapped into physical memory. This mapping process is managed by using page tables residing in RAM. To make use of the pages a cache in the memory management unit is used, called a Translation Lookaside Buffer (TLB). Once the page table reaches its maximum limit, performance is impacted due to cache misses [34]. Utilizing the Transparent HugePages support helps solve this problem.

Although THP originally is an optimization, it may negatively impact the performance of applications and services in certain cases. Database systems such as MongoDB and Splunk

are impacted by THP, where performance can be degraded as much as 30% according to Splunk [35].

File System Selection

Most modern Linux distributions use the EXT4 file system as default, but older machines that have not been reinstalled or updated in years may use older versions of the EXT file system. The older file systems do not have the same performance improvements that the newer file systems EXT4 and XFS have. While EXT3 and EXT4 are very similar, EXT4 supports larger files and has a higher performance for read and write operations.

The XFS file system can boost performance significantly when using faster drives such as solid state drives. For average systems, the differences in performance are minimal compared to EXT4. [36]

The differences in throughput in XFS and EXT4 can be seen in Figure 3.2.

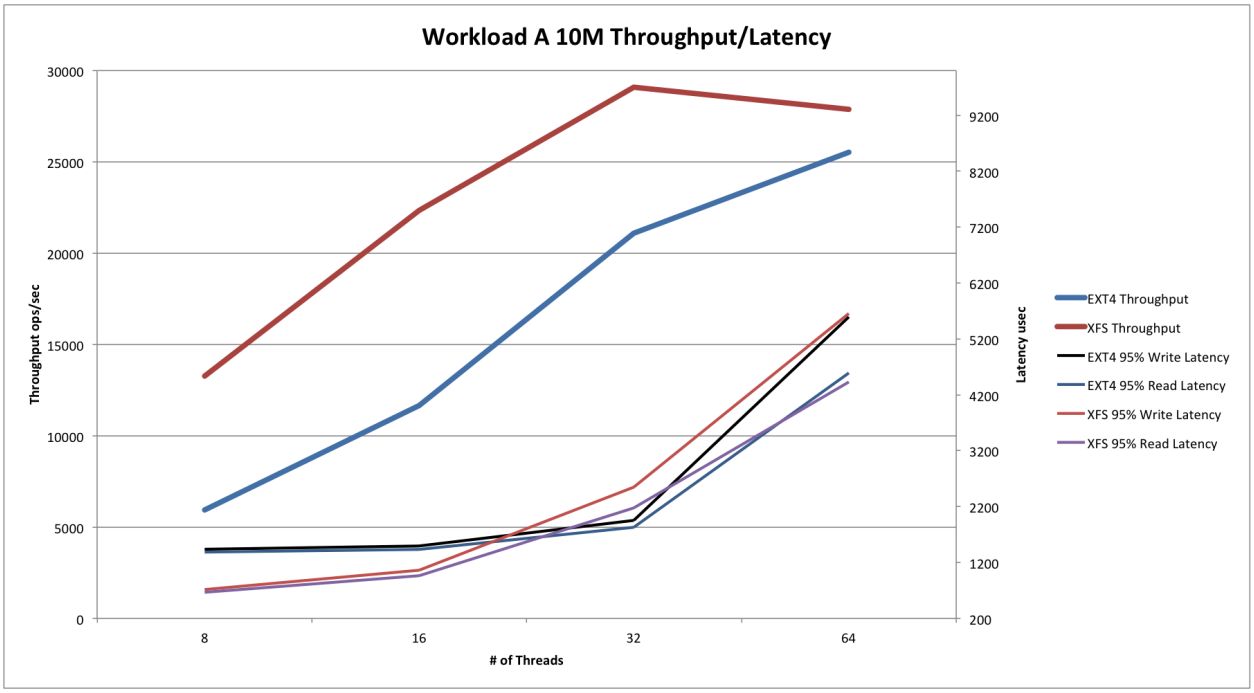


Figure 3.2: A comparison of the file systems EXT4 and XFS [36].

Input/Output Scheduler

In the Linux operating system, a scheduler handles input and output disk operations. The default I/O scheduler can be modified to provide better performance by altering settings in the Linux kernel [37].

According to research on scheduler optimizations, the *noop* scheduler provides the largest increase in performance [38].

3.2.6 Disk Space Optimization

Disk space optimization is not directly related to performance optimization but is included to provide a deeper understanding of how database systems store overhead information. Systems with low storage capacity can utilize this technique to reduce overhead.

When storing data in databases (i.e MongoDB with MMAPv1) using field names, excessive disk space is used the longer the field names are.

Storage space can be reduced by using shorter field names. For instance, *u* can be used as a field name instead of *username*. By using a field name consisting of 30 characters, one million empty documents with a field name of approximately 30 characters equal approximately 28MB. For 10 fields with similar length, 280 MB disk space is used to store one million empty documents [39].

3.3 Known Solutions for Big Data

Some database systems are specifically designed to handle big data. One of the most known solutions is Hadoop developed by Apache.

3.3.1 Apache Hadoop

Hadoop is a well-known collection of utilities for distributed database solutions developed by Apache. Hadoop distributes work to multiple nodes by utilizing the Map-Reduce functionality that organizes work in two tasks: Map tasks and reduce tasks between the available nodes.

In HDFS (Hadoop Distributed File System), the search engine Solr accounts for the searches of data and provides replication and distributed indexing in Hadoop clusters.

Apache Hadoop and Solr is a popular solution for distribution and scaling of extremely large amounts of data. However, Apache Hadoop is not the optimal solution for all problems due to the way Hadoop stores data. Each Hadoop node has a certain size, for instance 64 MB or 128 MB. For optimal performance, it is recommended that the Hadoop nodes are between 64 MB and 256 MB in size.

A Hadoop cluster may contain one hundred nodes like this. If the data to store is less than the maximum size of the node, the remaining space will be wasted as the nodes are of a fixed size. This is called the *small files problem*. For instance, storing a file of 2 MB in a 128 MB Hadoop node results in 126 MB wasted space. This makes it impossible to take advantage of HDFS as a file system if the data to be stored are multiple small files, resulting in possible excessive overhead. Additionally, Hadoop is not suitable for unique data.

Apache Hadoop is not included in the experimental testing in this thesis as the average user does not have multiple nodes available for a distributed setup, but has been included in this chapter as it is a state-of-the-art solution for handling big data.

3.3.2 Redis

Redis functions as an in-memory data store that persists on disk [40]. Even though Redis is a key-value store, it supports other values such as sets, lists, hashes, hashmaps, strings and more.

Based on existing research Redis is among the fastest database systems available in terms of performance but has not been included in the experimental tests in this research, as the

data sets are larger than the available memory resources.

3.4 Comparison Papers

Several articles exist on database system comparisons. Some of these provide a functional overview and comparison of available database systems, and some also include performance evaluations. The articles that are considered most important in regard to this thesis are presented in this section.

NoSQL Databases: MongoDB vs Cassandra

This paper describes the characteristics and features of NoSQL databases and operational principles. A comparison of the two NoSQL systems MongoDB and Cassandra is also performed, where run-times are evaluated [41].

The performance evaluation in the paper consists of multiple workloads and is tested with the Yahoo! Cloud Serving Benchmark service with up to 700.000 records.
Their results show that the performance of MongoDB decreases as the size of data increases, while the result is the opposite for Cassandra with a response time of 20 seconds for read queries on 700.000 records.

This thesis prioritizes the differences before and after applying optimization methods compared to the research in this paper.

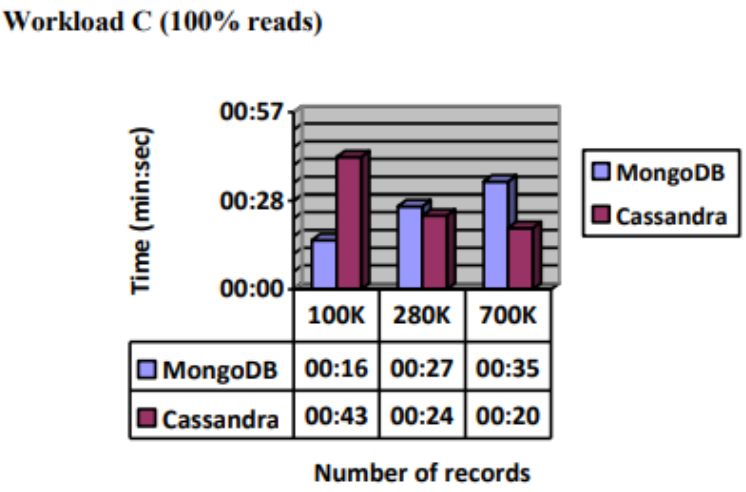


Figure 3.3: A comparison of read operations between MongoDB and Cassandra [41].

Experimental Evaluation of NoSQL Databases

A comparative study has been conducted of benchmarking 10 NoSQL database systems of three types; document-store, column stores, and key-value stores to compare the performance of each type. Elasticsearch and MongoDB are among the database systems that were evaluated. In the study, the researches utilize several workloads consisting of either read- or update operations, or a combination of the two and performs evaluations of load times and execution speeds using the Yahoo! Cloud Serving Benchmark service. All the systems were evaluated with 600.000 records of automatically generated data, consisting of 10 fields per record.

The overall results of their experimental testing can be seen in Figure 3.4.

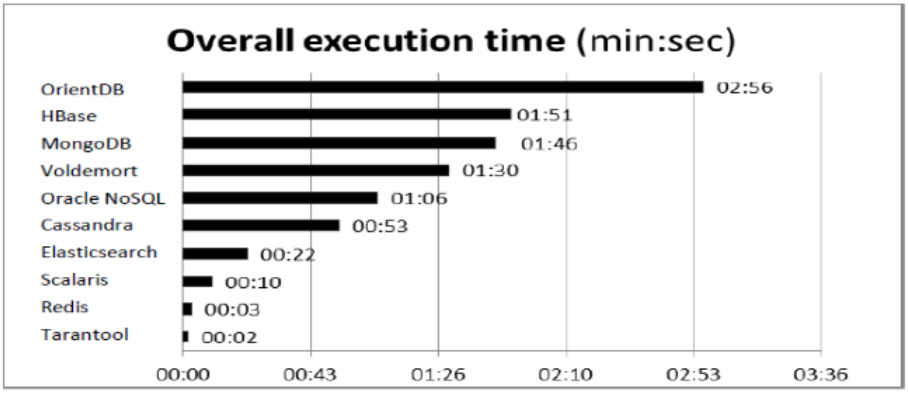


Figure 3.4: Overall execution time of 10 NoSQL database systems as presented in the article *Experimental Evaluation of NoSQL Databases* [15].

In comparison with other performance evaluation papers, the researchers of this paper do not evaluate throughput but instead measure performance by using query execution times. Their research confirms that in-memory storage engines perform better than other NoSQL database systems and specifically that key-value stores are superior to the competitors. The paper concludes that NoSQL database systems can perform well on huge data sets [15].

While the results in this research provide an overview of NoSQL performance, no optimization techniques were applied an in attempt to increase the performance to a greater extent, which is what distinguishes this research from this thesis.

A performance comparison of SQL and NoSQL databases

Yishan Li and *Sathiamoorthy Manoharan* have presented a performance comparsion of SQL and NoSQL databases, examining read, write, delete and instantiate operations.

The database systems tested in their research are MongoDB, RavenDB, CouchDB, Cassandra, Hypertable, Couchbase, and MicroSoft SQL Server Express.

While the two previous articles evaluated performance for a given number of records, this article uses the number of operations for performance evaluation.

Database	Number of operations					
	10	50	100	1000	10000	100000
MongoDB	8	14	23	138	1085	10201
RavenDB	140	351	539	4730	47459	426505
CouchDB	23	101	196	1819	19508	176098
Cassandra	115	230	354	2385	19758	228096
Hypertable	60	83	103	420	3427	63036
Couchbase	15	22	23	86	811	7244
MS SQL Express	13	23	46	277	1968	17214

TABLE II
TIME FOR READING (MS)

Figure 3.5: Results of read operations for SQL and NoSQL database systems from *A performance comparison of SQL and NoSQL databases* [42].

Their research concludes that Couchbase and MongoDB are the best performing solutions. If iterating through keys and values is not a requirement, Couchbase is the best solution, otherwise MongoDB is the best solution [42].

Comparing NoSQL MongoDB to an SQL DB

The paper *Comparing NoSQL MongoDB to an SQL DB* investigates the performance of NoSQL and SQL databases in terms of query execution time for a "modest sized database" [43].

The results of the experimental tests in this paper prove that MongoDB perform equally as good or even better compared to relational databases. The only exception is when utilizing aggregate queries [43].

The research concludes that MongoDB is a good candidate for larger data sets of non- or semi-structured data that does not require complex queries, otherwise a relational database should be used.

Summary

The existing research that is available on the thesis topic is not identical to the approach in this thesis, as none of the articles evaluate performance before and after optimizing each database system.

Chapter 4

Method

This chapter covers the installation and setup instructions for the database systems. To ensure fair and comparable results all solutions are set up and tested with equal computer hardware and specifications.

All database systems are installed using the newest stable version.

4.1 Environment

The database systems are installed in a virtualized environment using VMware ESXi version 6.7. VMware ESXi is a bare-metal hypervisor that allows a user to manage multiple virtual machines by partitioning and allocating resources on a server [44].

By using virtualization software an operating system can be installed on a virtual machine and snapshots of the current machine state can be created, and rolled back to at a later point if necessary. To prevent loss of data during a potential system failure, snapshots were created during the implementation to ensure backups of existing work.

Multiple tests are performed using different configuration settings and optimization methods. The testing stages are separated in the two following parts:

- **Default settings:** In the initial stage all database systems are installed and set up with default settings with no modifications.
- **Maximum optimization:** In this stage all possible optimization methods that are available to each database system is applied in an attempt to optimize performance as much as possible. Operating system optimizations are also utilized, along with index creation for MySQL and MongoDB.

The tests performed in this thesis are tested in an environment with hardware commonly used by regular users that do not have access to unlimited resources and nodes. This is done in an attempt to research performance optimization methods that can be useful for the average user.

4.2 Hardware & Specifications

- **Memory (RAM):** 20 GB DDR3 RAM @ 1600 MHz
- **Processors:** 2 x 2 cores - Intel Core i5 4690k @ 3.50 GHz
- **Storage:** Samsung 860 EVO 256GB & Seagate Firecuda 1.7 TB @ 7200 RPM
- **Operating System:** Debian 9, 64-bit
- **File system:** EXT4

4.3 Data Preprocessing & Insertion

The data focused on in this thesis originates from leaked databases from various websites. This data has different structures and formats. By using real data legal aspects apply related to obtaining and storing this information.

In order to avoid legal complications all data used in this thesis has been generated with a script that replicates the structure of real data found in publicly leaked databases.

4.3.1 Generating Data

The data generator script is written in the scripting language Python and generates usernames, emails, passwords, password hashes, salts and IP-addresses by the use of random functions. The script is implemented in such a way that a user can specify amount of records to be created and will generate data accordingly, writing to an output file.

This file can later be parsed by a data parser script that accounts for the insertion of data to the database systems. The data generator script can be found in appendix A.1.

4.3.2 Python File Parser

To insert data into the database systems a Python script has been created that takes a data file and a regular expression as input. The script reads the data file by line and stores data in memory in chunks of x size(lines). Once the specified chunk size has been reached, the data is inserted to the selected database unless a *dry run* is performed. This process is repeated until the full file has been parsed. The script can be found in appendix A.2.

Parser usage syntax:

```
1 Example usage: python3 parser.py leaked-data-1m.txt master_leak_1m
    ↪ 10000 elasticsearch 127.0.0.1 9200
2 Input regex: (?P<username>.*)\,(?P<email>.*)\,(?P<password>.*)\,(?P<
    ↪ password_hash>.*)\,(?P<salt>.*)\,(?P<ip>.*)
3 Perform dry run? [y/n]: n
```

Full source code with modules can be obtained from Github [45].

4.4 Installation & Setup

Installation and setup instructions for the database systems are described in this section. The optimization methods used are also included in a subsection for each database system.

4.4.1 MySQL Percona

Default Installation

The latest version of Percona server for MySQL is version 8.0.15-5. Installation pacakges can be downloaded from the official website of Percona [46].

Percona offers downloads of separate packages, but also a bundled installation package containing all parts of the Percona MySQL server.

The following commands were executed to download, extract and install Percona server for MySQL:

```
1 # wget https://www.percona.com/downloads/Percona-Server-LATEST/Percona-
    ↪ Server-8.0.15-5/binary/debian/stretch/x86_64/Percona-Server
    ↪ -8.0.15-5-rf8a9e99-stretch-x86_64-bundle.tar
2 # tar -xvf Percona-Server-8.0.15-5-rf8a9e99-stretch-x86_64-bundle.tar
3 # dpkg -i *percona-*.deb
4 # apt -f install
```

Upon successful installation the MySQL server can be started by issuing the command:

sudo service mysql start

To secure the default installation and set a root password for management of the database system, the following command is used:

/usr/bin/mysql_secure_installation

At this point the Percona MySQL server is installed with default settings and is ready for testing.

Optimization

The majority of the performance optimizations for MySQL Percona are modifications made to variables in the configuration file of MySQL, specifically related to the storage engine XtraDB. XtraDB is an improved version of the original storage engine in MySQL, InnoDB.

Although changes to the MySQL server are made for improved performance as new updates are released, it is still possible to further improve the server in accordance with available hardware on the host machine to reach peak performance.

Alexander Rubin has published results from research in his post titled *MySQL 5.7 Performance Tuning Immediately After Installation*. This post describes optimizations that can be applied by altering the default values of a MySQL version 5.7 installation [47]. As MySQL version 8 is the successor of MySQL version 5.7, many of the same configuration variables are still used.

MySQL Configuration

Multiple variables has been added or modified in the MySQL configuration file for the optimization process. The most important variables are the following:

innodb_buffer_pool_size: This variable controls the size of the MySQL buffer and specifies the limitation of maximum usable memory. This variable has been set to 16G, and is recommended to be between 80-90% of available memory.

innodb_buffer_pool_instances: This variable controls the maximum number of buffer instances that can be created and has been set to a value of 8. With a buffer pool size of 16G, each buffer pool instance can use 2GB of memory each, which is twice as much as the minimum limit of 1GB per instance.

Parts of the changes made to the confguration file of MySQL is obtained from a Benchmarking post by the co-founder of Percona, *Vadim Tkachenko* [48]. The full configuration file for MySQL can be found in appendix A.3.

Operating System Changes

To tune the operating system the following changes were applied:

- Disable SWAP memory:

```
sudo swapoff -a
```

- Change IO disk scheduler in Linux to *noop* as described in Section 3.2.5:

```
sudo echo noop > /sys/block/sdb/queue/scheduler
```

- Adjust maximum locked memory (-l) and open files (-n) in the Linux kernel limits (temporarily):

```
ulimit -n 65535
ulimit -l unlimited
```

- Alternatively, limits can also be set permanently by adding entries to */etc/security/limits.conf* as follows:

```
mysql       soft        nofile      65535
mysql       hard        nofile      65535
mysql       soft        nproc       65535
mysql       hard        nproc       65535
```

- Reload the system control daemon and MySQL service:

```
sudo systemctl daemon-reload
sudo service mysql restart
```

Index Creation

For the second part of the optimization for MySQL indexes have been created to ensure fair testing results as two of the other systems use indexing by default.

Typical searchable fields are username, email, password, password_hash and ip. InnoDB supports B-Tree indexing. Single indexes of this type were created for each of the searchable fields. Initially a compound index was created for all fields, but did not perform as well as single indexes.

Single indexes were created by executing the query:

```
1 CREATE INDEX index_name ON table_name (column_list)
```

4.4.2 MongoDB Percona

Default Installation

The latest version of Percona server for MongoDB is version 4.0.9-4. Similar to Percona MySQL, the installation packages can be downloaded from the official website of Percona [49].

Percona offers downloads of separate packages, but also a bundled installation package containing all parts of the Percona MongoDB Server.

The following commands were executed to download, extract and install Percona server for MongoDB:

```
1 # wget https://www.percona.com/downloads/percona-server-mongodb-LATEST/
    ↪ percona-server-mongodb-4.0.9-4/binary/debian/stretch/x86_64/
    ↪ percona-server-mongodb-4.0.9-4-r2b2d452-stretch-x86_64-bundle.tar
2 # tar -xvf percona-server-mongodb-4.0.9-4-r2b2d452-stretch-x86_64-
    ↪ bundle.tar
3 # dpkg -i *percona-server*.deb
4 # apt -f install
```

Once the packages finish installing, remote access to the MongoDB server can be enabled by adjusting the MongoDB configuration file. This is optional, and can be done by modifying */etc/mongod.conf* and replacing the bindIp accordingly to enable remote access from any IP address:

```
1 # network interfaces
2 net:
3   port: 27017
4   bindIp: 0.0.0.0
```

After performing modifications the MongoDB service can be started using the command: *sudo service mongod start*

At this point the Percona MongoDB server is installed with default settings and is ready for testing.

Optimization

Modifications to the configuration file of MongoDB is quite limited compared to MySQL, with a small number of configurable variables for optimization. The variables that can be changed depends on the selection of storage engine in MongoDB, with three options; MMAPv1, WiredTiger and In-memory storage.

There are pros and cons with each storage engine type in MongoDB. In the most recent releases of MongoDB, WiredTiger offers increased concurrency and other performance improvements that makes it faster than MMAPv1. As MMAPv1 also is deprecated from MongoDB 4.0 and the data to be stored exceeds the available size of memory, WiredTiger is selected as storage engine.

Due to the minor changes that possibly can be made to the MongoDB configuration file, primarily operating system changes are made in attempts to improve the performance of MongoDB.

WiredTiger Configuration

cacheSizeGB: This variable controls the maximum usable memory for MongoDB. Similar to the configuration of MySQL this variable is set to 16GB.

journalCompressor & blockCompressor: In WiredTiger two compression methods are available; *snappy* and *zlib*. Snappy is the default method and the fastest of the two. This value is left at its default value as saving disk space is not relevant for this thesis.

prefixCompression: Prefix compression is also left enabled (default) because it reduces the memory consumed by the indexes to a great extent. This allows more memory to be used for document storage or other indexing or search jobs, ultimately resulting in improved performance.

The full MongoDB configuration file can be found in appendix A.4.

Operating System Changes

To tune the operating system the following changes were applied:

- Disable SWAP memory:

```
sudo swapoff -a
```

- Change IO disk scheduler in Linux to *noop* as described in Section 3.2.5.

```
sudo echo noop > /sys/block/sdb/queue/scheduler
```

- Adjust maximum locked memory (-l) and open files (-n) in the Linux kernel limits (temporarily):

```
ulimit -n 65535
ulimit -l unlimited
```

- Alternatively, limits can also be set permanently by adding entries to */etc/security/limits.conf* as follows:

```
mongod        soft        nofile        65535
mongod        hard        nofile        65535
mongod        soft        nproc         65535
mongod        hard        nproc         65535
```

- Since the operating system used is Debian, Transparent HugePages can be disabled for MongoDB:

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

- Reload the system control daemon and MongoDB service:

```
sudo systemctl daemon-reload
sudo service mongod restart
```

Index Creation

Similar to MySQL, MongoDB do not utilize indexing by default. In MongoDB it is possible to create sparse indexes for data. This feature is typically used when data does not have all fields set and could be beneficial when dealing with real leaked data from various sources due to the differences in structure and data. As all fields are set by the data generator, single binary tree indexes were created for each of the searchable fields; username, email, password, password_hash and ip.

Single indexes were created by executing the query:

```
1 db.getCollection('collection').createIndex({'field': 1});
```

4.4.3 Elasticsearch

Default Installation

The current stable version of Elasticsearch is 6.6.1. This version of Elasticsearch was installed by following the guidelines by the Elastic team [50].

The Elasticsearch PGP Key was imported to the package manager in Debian, and Elasticsearch installed using apt:

```
1 # wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo
    ↪ apt-key add -
2 # sudo apt-get install apt-transport-https
3 # echo 'deb https://artifacts.elastic.co/packages/6.x/apt stable main'
    ↪ | sudo tee -a /etc/apt/sources.list.d/elastic-6.x.list
4 # sudo apt-get update && sudo apt-get install elasticsearch
```

Upon successful installation, Elasticsearch can be started with the command:

sudo service elasticsearch start

At this point Elasticsearch is installed with default settings and is ready for testing.

Elasticsearch serves a HTTP API operating over TCP port 9200 that is used for insertion and management of data. The Python file parser operates with this API for inserting data. For indexing purposes the number of replicas was set to 0 in the default experimental testing and is the only setting that is changed from its default value.

According to the Elasticsearch developers virtualization is not recommended as Elasticsearch performs better in non-virtualized environments where it can provide an increase in performance. For this thesis all systems are tested in virtualized environments to ensure fair results.

It is also recommended that each Elasticsearch node is allocated no more than 50% of the available memory.

Optimization

Elasticsearch utilizes multiple configuration files where settings can be adjusted for the Java Virtual Machine that runs the service.

The configuration files can be found in */etc/elasticsearch/* where the files *elasticsearch.yml* and *jvm.options* are made changes to in attempts of improving performance.

Elasticsearch Configuration

- Allocate more memory to the Elasticsearch heap by modifying the file */etc/elasticsearch/jvm.options* and increasing heap size to approximately 50% of total available memory as suggested by Elasticsearch developers:

```
1 # Xms represents the initial size of total heap space
2 # Xmx represents the maximum size of total heap space
3
4 -Xms10g
5 -Xmx10g
```

- Lock the memory to preserve it for Elasticsearch only by modifying */etc/elasticsearch/elasticsearch.yml* accordingly:

```
1 bootstrap.memory_lock: true
```

Operating System Changes

To tune the operating system the following changes were applied:

- Disable SWAP memory:

```
1 sudo swapoff -a
```

- Adjust maximum locked memory (-l) and open files (-n) in the Linux kernel limits (temporarily):

```
ulimit -n 65535
ulimit -l unlimited
```

- Alternatively, limits can also be set permanently by adding entries to */etc/security/limits.conf* as follows:

```
elasticsearch soft memlock unlimited
elasticsearch hard memlock unlimited
elasticsearch soft nofile 65535
elasticsearch hard nofile 65535
```

- Increase map count to avoid memory exceptions for index storage:

```
sysctl -w vm.max_map_count=262144
```

- Override default Elasticsearch memory limitations by using the command *sudo systemctl edit elasticsearch*, creating the file */etc/systemd/system/elasticsearch.service.d/override.conf*, and adding the content:

```
[Service]
LimitMEMLOCK=infinity
```

- Reload the system control daemon and Elasticsearch service:

```
sudo systemctl daemon-reload
sudo service elasticsearch restart
```

When evaluating performance of the optimized version of Elasticsearch, one replica shard was used. Additional replica shards were not used as the virtual machine only has two processors available.

4.4.4 Splunk

Default Installation

Splunk is the only semi-enterprise solution tested and offers a 30 day free enterprise trial. The latest stable version of Splunk is v7.2.5.1 which was downloaded after creating a user at Splunk.com [51].

The following commands were executed to download and install Splunk:

```
1 # wget -O splunk-7.2.5.1-962d9a8e1586-linux-2.6-amd64.deb 'https://www.
    ↪ splunk.com/bin/splunk/DownloadActivityServlet?architecture=x86_64
    ↪ &platform=linux&version=7.2.5.1&product=splunk&filename=splunk
    ↪ -7.2.5.1-962d9a8e1586-linux-2.6-amd64.deb&wget=true'
2 # dpkg -i splunk-7.2.5.1-962d9a8e1586-linux-2.6-amd64.deb
```

Splunk comes with an automated installer and can upon completion be started with the command:

sudo /opt/splunk/bin/splunk start

After starting the service Splunk prompts the user to enter new credentials used for managing Splunk through the Splunk web interface that is available over TCP port 8000.

At this point, Splunk is installed with default settings and ready for testing.

In comparison with the other database systems Splunk has the ability to read directly from local file storage. The Python parser was therefore not required for inserting data in this database system.

Optimization

Performance optimization methods specified in the Splunk documentation include either horizontal scaling or using multiple indexes [52]. Certain configuration edits can also be made although Splunk recommend using the default values.

In comparison with the optimization of the other database systems there is no memory limitation for Splunk, it will therefore use all available resources if necessary.

By default Splunk uses the configuration files residing in *$SPLUNK_HOME/etc/system/default* but can be overridden by creating configuration files in *$SPLUNK_HOME/etc/system/local* for the local instance of Splunk.

The most significant search head performance optimization methods available have been applied as suggested in two conference presentations on Splunk performance tuning [53] [54].

Splunk Configuration

- Enable batch mode search parallelization. Batch mode parallelization is designed to search through data in buckets instead of per event [55]. By setting this value to 2, input/output, processing and memory operations are multiplied in batch mode searches. According to Splunk documentation a value of 2 provide the best performance increase [55] and faster retrieval of results [54]:

```
1 [search]
2 allow_batch_mode = true
3 batch_search_max_pipeline = 2
```

- Set maximum hot buckets for indexes in */opt/splunk/etc/system/indexes.conf* and increase maximum bucket data size. The number of buckets searched impacts search performance:

```
1 [default]
2 maxHotBuckets = 10
3 maxDataSize = auto_high_volume
```

- Configure Parallel summarization in */opt/splunk/etc/system/savedsearches.conf* [56]:

```
1 [default]
2 auto_summarize.max_concurrent = 2
```

- Configure index parallelization in */opt/splunk/etc/system/server.conf* [56]:

```
1 [general]
2 ...
3 parallelIngestionPipelines = 2
```

Operating System Changes

- As described in Section 3.2.5, Transparent HugePages can be disabled, possibly resulting in a 30% performance improval in Splunk:

```
1 echo never > /sys/kernel/mm/transparent_hugepage/enabled
2 echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

Verify that Transparent HugePages are not used by Splunk:

```
1 # grep hugetables /opt/splunk/var/log/splunk/splunkd.log
2 05-07-2019 15:50:55.240 +0200 INFO  ulimit - Linux transparent
    ↪ hugepage support, enabled="never" defrag="never"
3 05-07-2019 15:52:46.291 +0200 INFO  ulimit - Linux transparent
    ↪ hugepage support, enabled="never" defrag="never"
```

- Reload the system control daemon and Splunk service:

```
1 sudo systemctl daemon-reload
2 sudo /opt/bin/splunk restart
```

Part III

Experiments & Results

Chapter 5

Experiments

This chapter presents the experimental results based on the implementation from chapter 4. The results are separated in two sections as there are two implementations. First the performance results from the selected database systems running with default settings are presented. Thereafter the performance results of the same systems running with optimized settings. Finally, a comparison of all systems is presented and vizualized to provide a better understanding of the differences in results.

5.1 Test Cases

For each stage the database systems' performance is evaluated with a series of records starting at 1 million records, 5 million records, 10 million records, 50 million records, ending at 100 million records. All results are documented in tables and graphs containing the number of records queried and response times for both regular and wildcard[1] queries.

As the two systems MySQL Percona and MongoDB Percona are not using indexes by default, an extra section of results is included to ensure fair comparisons to the other systems that utilize indexes by default.

5.2 Database System Results

During the experimental testing some of the database systems cached the query results temporarily, resulting in faster query response times for queries that already had been executed. Due to this, new queries were made for each test. The final results are based on combined queries of both exact and wildcard searches with an average of 5-10 query responses per test.

5.2.1 MySQL Results

The optimizations made to the MySQL configuration and operating system tuning made minor impact on the results when the amount of records were below 10 million but made a larger impact on larger data sets. A significant increase was seen at 50+ million records for both exact and wildcard queries with the specified hardware settings.

After creating indexes MySQL delivered query responses within milliseconds and accounts for the largest improvement, indicating the importance of utilizing indexes for fast searches.

The performance evaluation of MySQL can be seen in Table 5.1, and has been visualized in Figure 5.1.

[1] A wildcard operator allows a user to search for sub-strings within a record, such as querying "hello*" would match all strings that begins with "hello".

Records	Def_{Exact}	$Def_{Wildcard}$	Opt_{Exact}	$Opt_{Wildcard}$	Opt_{Exact_i}	$Opt_{Wildcard_i}$
1.000.000	0.46 sec	0.52 sec	0.38 sec	0.39 sec	0.015 sec	0.073 sec
5.000.000	2.03 sec	2.64 sec	1.69 sec	1.73 sec	0.026 sec	0.307 sec
10.000.000	4.01 sec	4.28 sec	3.43 sec	3.53 sec	0.031 sec	0.606 sec
50.000.000	20.78 sec	26.20 sec	17.55 sec	18.58 sec	0.033 sec	13.295 sec
100.000.000	172.50 sec	174.50 sec	150.90 sec	159.08 sec	0.038 sec	23.932 sec

Table 5.1: Query results from default settings, optimized settings and optimized settings with indexes for MySQL.

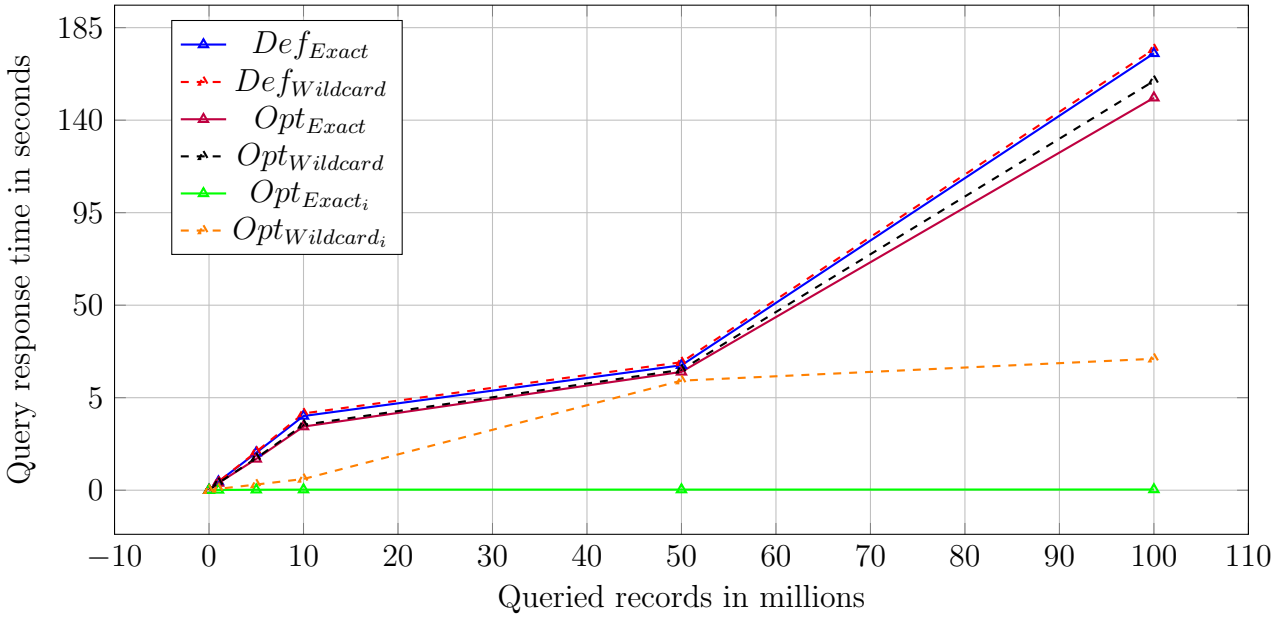


Figure 5.1: Results from performance evaluation of Percona Server for MySQL.

5.2.2 MongoDB Results

Minor changes are seen after applying optimizations to MongoDB. This is not surprising as the configuration files do not offer many possibilities for tuning.

Some tuning approaches resulted in performance worse than the default configuration. A minimal amount of variables was therefore changed in the final configuration used in the evaluation process.

Similar to MySQL there was a minimal performance increase from the configuration optimization and indexing accounted for the best improvements.

Greater results could be obtained by using the in-memory storage engine of MongoDB but this is not a viable solution when dealing with a large amount of leaked data.

By comparing the graphical results of MySQL and MongoDB, similarities are seen between the systems although they follow different database models.

The performance evaluation of MongoDB can be seen in Table 5.2 and has been visualized in Figure 5.2.

Records	Def_{Exact}	$Def_{Wildcard}$	Opt_{Exact}	$Opt_{Wildcard}$	Opt_{Exact_i}	$Opt_{Wildcard_i}$
1.000.000	0.34 sec	0.45 sec	0.32 sec	0.42 sec	0.015 sec	0.151 sec
5.000.000	1.65 sec	2.16 sec	1.61 sec	2.12 sec	0.015 sec	0.697 sec
10.000.000	3.31 sec	4.49 sec	3.22 sec	4.24 sec	0.016 sec	1.377 sec
50.000.000	42.81 sec	48.58 sec	41.36 sec	48.48 sec	0.016 sec	7.560 sec
100.000.000	179.40 sec	180.66 sec	168.16 sec	177.28 sec	0.024 sec	15.240 sec

Table 5.2: Query results from default settings, optimized settings and optimized settings with indexes for MongoDB.

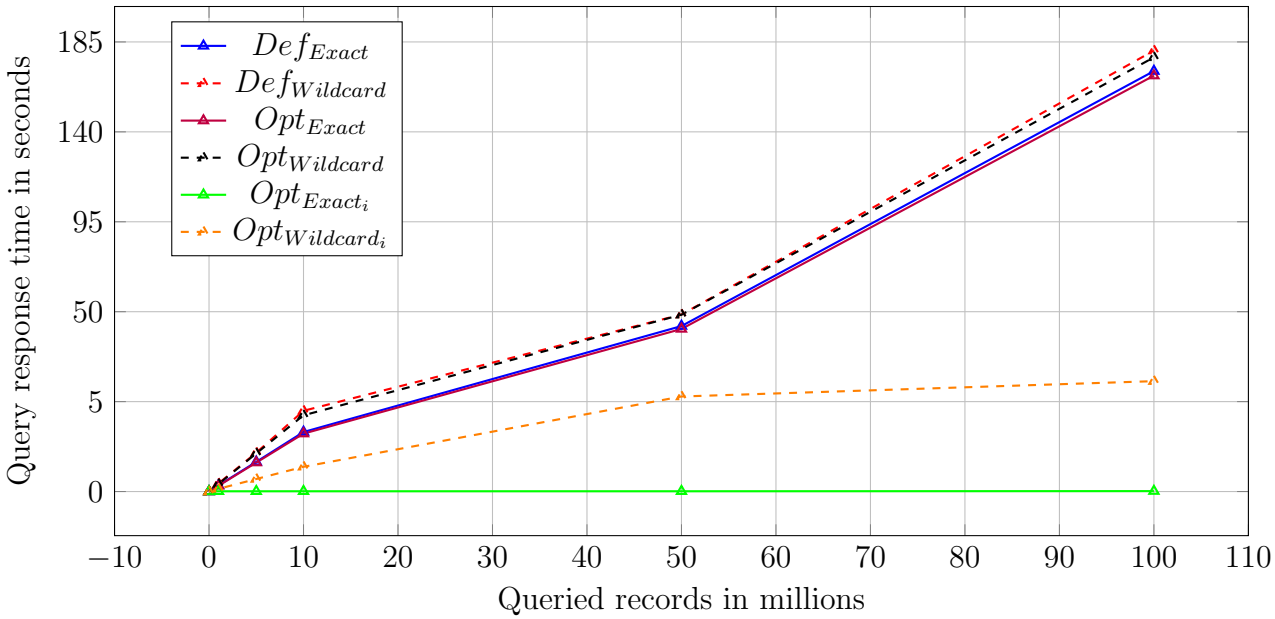


Figure 5.2: Results from performance evaluation of Percona Server for MongoDB.

5.2.3 Elasticsearch Results

The results of testing Elasticsearch were impressive due to the rapid response times. The reason behind the rapid response times is that Elasticsearch utilizes the inverted index structure described in Chapter 3, Section 2.4.

Elasticsearch uses analyzers that translate data when indexing. For instance, the default analyzer will translate characters into lowercase where i.e the word "*Foo BAR*" will be indexed as "*foo, bar*".

Wildcard queries are not analyzed which means the queried input is compared exactly to a term in the inverted index. Therefore the query must be converted to lowercase before being processed. A wildcard query for **BAR** will not match, since *BAR* does not equal *bar*. To solve this problem other analyzers such as *ngram* or *edge ngram* can be used according to Elasticsearch developer David Pilato [57].

The default testing did include the use of replica shards. In the optimized testing one replica shard was used, which resulted in four times as fast response times. This was discovered by accident when testing 10 million records versus 50 million records, where faster results were achieved on 50 million records as this was the only index that used replica shards. This number may not be the optimal number of shards, as the shard amount depends on the amount of nodes in the Elastic cluster.

Compared to the default installation there was a small increase in performance when using the optimized settings. Based on the results there is no need to further tune Elasticsearch for faster response times.

The performance evaluation of Elasticsearch can be seen in Table 5.3 and has been visualized in Figure 5.3.

Records	**Def.** $Q_{Regular}$	**Def.** $Q_{Wildcard}$	**Opt.** $Q_{Regular}$	**Opt.** $Q_{Wildcard}$
1.000.000	0.017 sec	0.068 sec	0.012 sec	0.060 sec
5.000.000	0.260 sec	0.289 sec	0.166 sec	0.276 sec
10.000.000	0.384 sec	0.863 sec	0.322 sec	0.682 sec
50.000.000	0.591 sec	1.176 sec	0.477 sec	0.840 sec
100.000.000	0.852 sec	2.506 sec	0.792 sec	2.370 sec

Table 5.3: Query results from default and optimized settings for Elasticsearch.

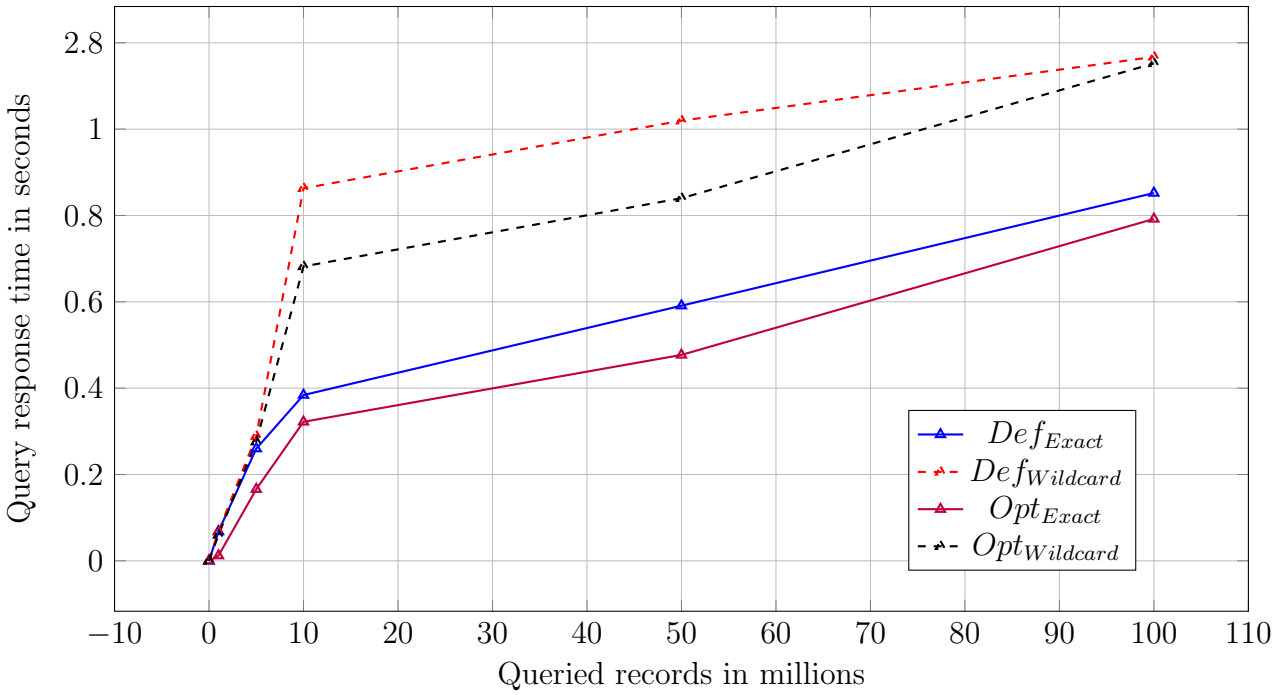


Figure 5.3: Results from performance evaluation of Elasticsearch.

5.2.4 Splunk Results

Splunk provided good results in the experimental testing but was not as fast as Elasticsearch.

Research shows that the optimizations applied to Splunk made a great impact on the performance of wildcard queries, where the response time was reduced with up to 75.5% compared to default settings.

Splunk's data storage uses self developed technology for indexing and is similar to the way an inverted index stores terms and document id's [58]. During experimental testing it was discovered that fields such as IP-address were slow because of the way full-text searches work. By using the TERM() operator in a Splunk query for IP-addresses, the full input is matched instead of tokens, resulting in a faster query response [59].

While Splunk is a solid solution for indexing and management of data, it is not the best solution for fastest possible query response times in large amounts of leaked data but offers great functionality for indexing and searches in log data with timestamps.

The performance evaluation of Splunk can be seen in Table 5.4 and has been visualized in Figure 5.4.

Records	**Def.** $Q_{Regular}$	**Def.** $Q_{Wildcard}$	**Opt.** $Q_{Regular}$	**Opt.** $Q_{Wildcard}$
1.000.000	0.28 sec	0.320 sec	0.137 sec	0.173 sec
5.000.000	0.53 sec	1.610 sec	0.441 sec	0.584 sec
10.000.000	0.88 sec	3.340 sec	0.719 sec	0.920 sec
50.000.000	1.35 sec	15.10 sec	1.110 sec	4.710 sec
100.000.000	2.40 sec	27.69 sec	1.753 sec	6.787 sec

Table 5.4: Query results from default and optimized settings for Splunk.

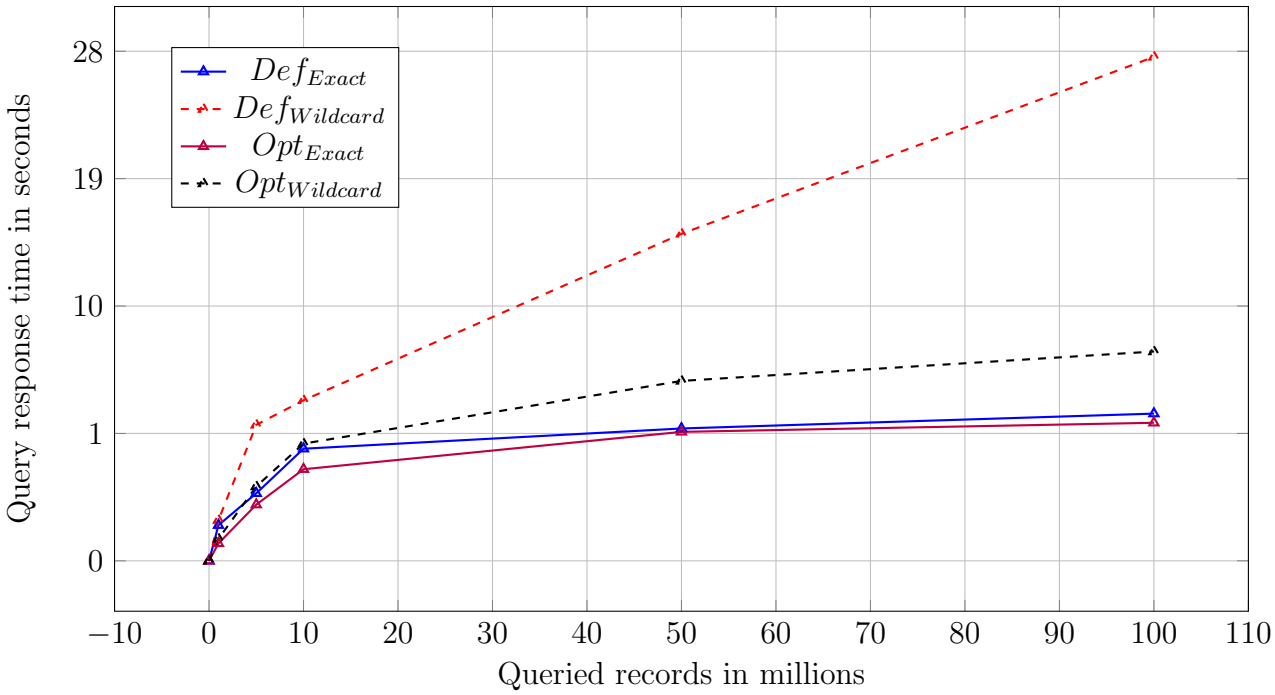


Figure 5.4: Results from performance evaluation of Splunk.

5.3 Comparison

In this section the results from the two testing stages are combined (exact- and wildcard queries) and the database systems compared to present the best possible solution.

5.3.1 Default Results

The default results show that MySQL and MongoDB performs equally for 10 million records or less. At 50 million records the performance of both systems start decreasing while the search engines Elasticsearch and Splunk handle larger data sets more efficiently.

Elasticsearch delivered the best performance on default installation with a response time of 1.68 seconds on 100 million records. MongoDB was the slowest with a result of 180 seconds on 100 million records, while MySQL was slightly faster, using 173.5 seconds. The response time in Splunk scaled according to the amount of records queried, and used 15 seconds to search through 100 million records.

The performance evaluation of all systems with default settings can be seen in Table 5.5 and has been visualized in Figure 5.5.

Records	MySQL	MongoDB	Elasticsearch	Splunk
1.000.000	0.46 sec	0.39 sec	0.043 sec	0.30 sec
5.000.000	2.05 sec	1.90 sec	0.275 sec	1.07 sec
10.000.000	4.08 sec	3.90 sec	0.623 sec	2.11 sec
50.000.000	21.52 sec	45.69 sec	0.883 sec	8.22 sec
100.000.000	173.50 sec	180.03 sec	1.679 sec	15.0 sec

Table 5.5: Query results from all systems using default settings and combined queries.

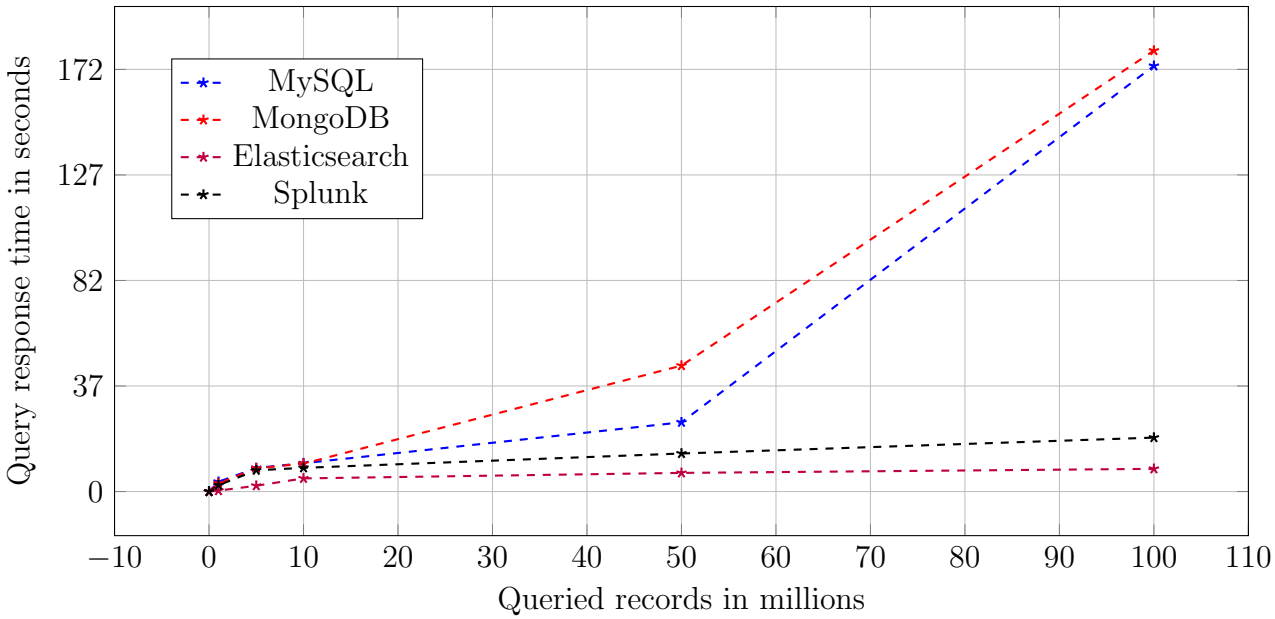


Figure 5.5: Results of search queries with default settings.

5.3.2 Optimized Results

The optimized results are similar to the default results. Research shows that the optimization methods specified in chapter 4 made a minor impact on the database systems Elasticsearch, MongoDB and MySQL. Splunk was the system most impacted by the performance optimizations applied, and resulted in up to 75.5% decrease in response time. Utilizing operating system optimization methods and configuration modifications, all database systems achieved a response time of less than 4 seconds for 10 million records.

The performance evaluation of all systems with optimized settings can be seen in Table 5.6 and has been visualized in Figure 5.6.

Records	MySQL	MongoDB	Elasticsearch	Splunk
1.000.000	0.39 sec	0.37 sec	0.036 sec	0.155 sec
5.000.000	1.71 sec	1.86 sec	0.221 sec	0.512 sec
10.000.000	3.49 sec	3.73 sec	0.502 sec	0.819 sec
50.000.000	18.07 sec	44.92 sec	0.658 sec	2.910 sec
100.000.000	154.99 sec	172.72 sec	1.581 sec	4.270 sec

Table 5.6: Query results from all systems using optimized settings and combined queries.

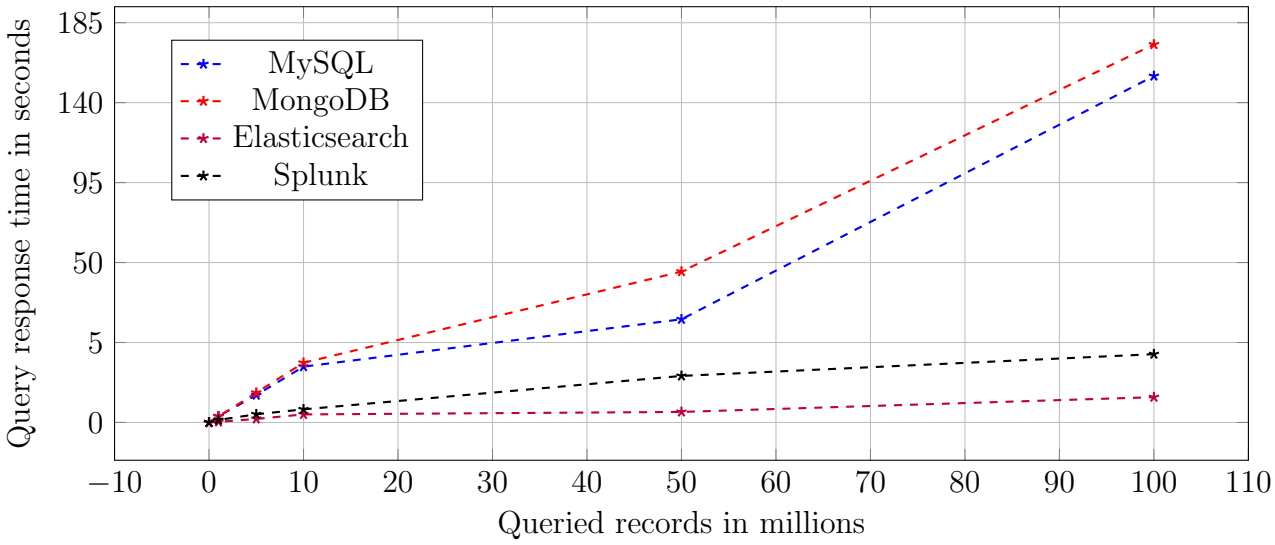


Figure 5.6: Results of search queries with maximum optimization options.

5.3.3 Optimized Results With Indexes

The final results show that MySQL and MongoDB can be optimized with indexing to perform better than the other systems for exact queries. For wildcard queries the two performs poorly. An interesting observation is that the response time in both MySQL and MongoDB seemed to increase significantly after 50 million records with the current hardware specifications.

As seen in Figure 5.8 the database systems Splunk, MongoDB and MySQL delivers identical performance at 30 million records. If the data set consists of 10 million records or less MySQL delivers the fastest query response times for combined queries as seen in Figure 5.8.

Our research confirms that Elasticsearch overall delivers the best performance in terms of response time in all tests for searching leaked data.

The performance evaluation of all systems with optimized settings and indexes can be seen in Table 5.7 and has been visualized in Figure 5.7 and Figure 5.8.

Records	**MySQL**	**MongoDB**	**Elasticsearch**	**Splunk**
1.000.000	0.044 sec	0.083 sec	0.036 sec	0.155 sec
5.000.000	0.166 sec	0.356 sec	0.221 sec	0.512 sec
10.000.000	0.319 sec	0.696 sec	0.502 sec	0.819 sec
50.000.000	6.664 sec	3.788 sec	0.658 sec	2.910 sec
100.000.000	11.985 sec	7.632 sec	1.581 sec	4.270 sec

Table 5.7: Query results from all systems using optimized settings with indexes and combined queries.

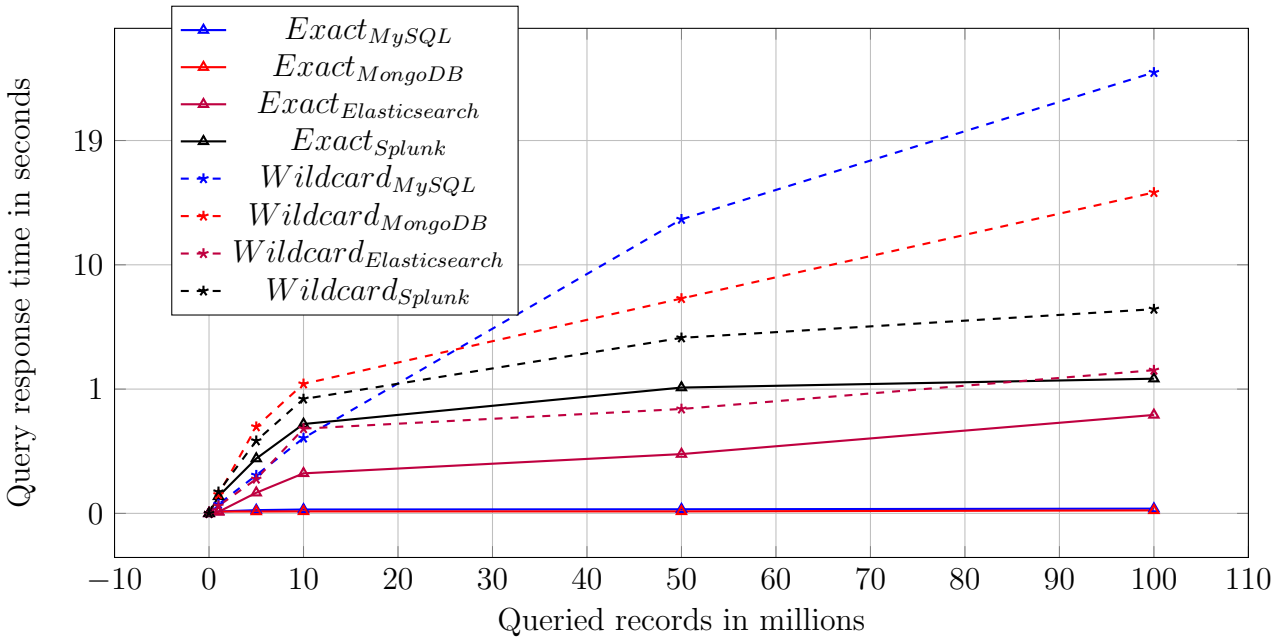


Figure 5.7: Results for exact and wildcard search queries using maximum optimization options and indexes.

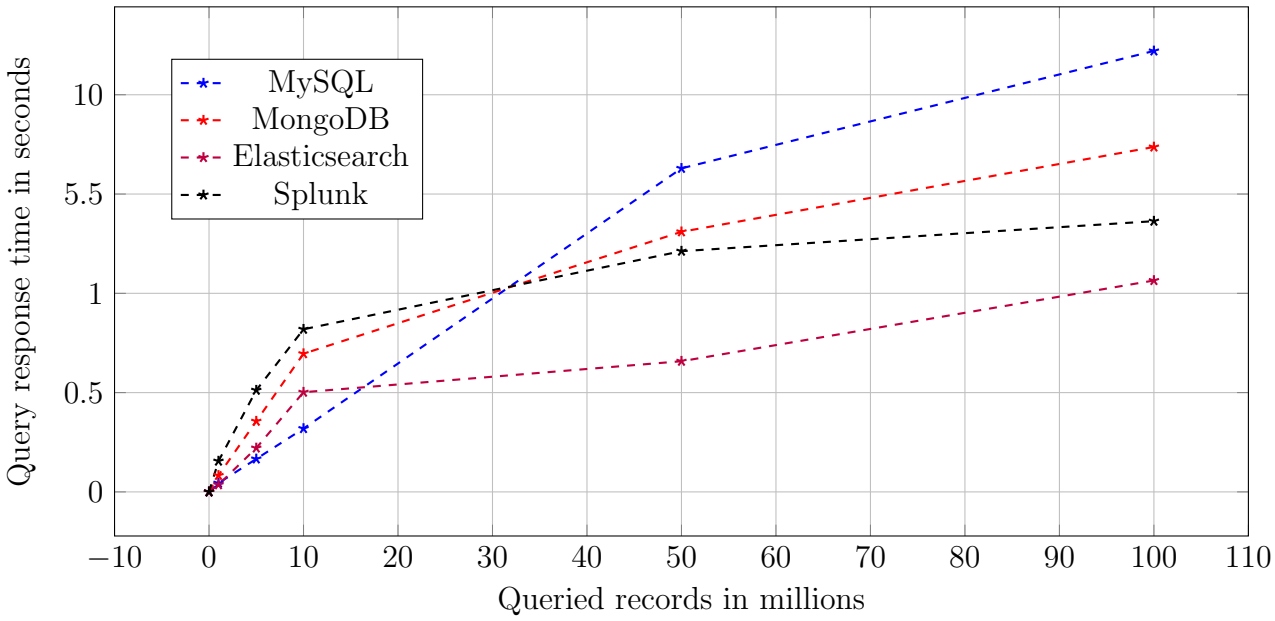


Figure 5.8: Results for combined search queries using maximum optimization and indexes.

5.4 Discussion

The research in this thesis attempts to find the best solution with concerning performance when querying large amounts of leaked data using average hardware and a single node.

The experimental tests performed provided both expected and unexpected results in terms of performance and confirms that several optimization techniques can be used to increase the query response time in database systems. For some of the database systems, the optimization techniques made a large impact on the results. As four database systems with three different structures were tested against each other, the results showed distinguishable differences.

MySQL and MongoDB performed almost equally as none of these databases index data by default, although they are based on different database models. The search engines Elasticsearch and Splunk also delivered results that did not differ greatly from each other.

Data Structure

In NoSQL database systems the database model is specifically built based on what the user wants to search for. Leaked data can originate from multiple web sites that are using different structures. The fields within the leaked data are normally similar to a certain extent, making leaked data suitable for both relational and NoSQL databases.

Using the data generator the most common fields found in leaked data are generated; username, email, password, password_hash, salt, and IP-address. Since all fields are populated using the data generator, the data is considered structured. Real leaked data, on the other hand, does not always include all fields but very often the same fields, making it semi-structured. Although there is a minor difference between generated and real data, the results are the same for search performance.

It is possible that the results of the experimental testing not necessarily will be identical for larger amounts of data or other types of data. The result of this thesis gives a general indication of which performance optimization methods can be used to achieve faster query response times.

Distribution and Sharding

Even better results could have been obtained by using a cluster of multiple distributed nodes in addition to sharding. An accidental mistake when testing Elasticsearch proved that using

one replica set (shard) resulted in four times as fast query response times. This is not necessarily the optimal number of replica shards but can be adjusted. Adjusting this setting is most often used when there is more than one machine available for computational processing, such as clusters.

Selection of disks

In the experimental testing, solid state drives (SSD) were used for storage initially but were replaced with solid state hybrid drives (SSHD) due to the large amount of data. According to research on Elasticsearch optimization and usage of disks, using solid-state drives deliver five times as fast performance as hard disk drives [60]. Using SSDs is therefore one of the best performance optimizations available as these disks deliver "near maximum performance at a fraction of the cost of an equivalent RAM solution" [61].

Wildcards

For wildcard searches and searches without indexing the entire database must be scanned to find a match for the query and is a time-consuming process [62]. Some wildcard searches are faster than others. A wildcard search querying *Starts with... (i.e: hello*)* can use indexes when searching, while queries such as *Contains... (i.e. he*lo)* and *Ends with... (*ello)* takes significantly longer. This depends on the type of database system and the way each system utilizes the indexes for wildcard searches. MongoDB and MySQL perform poorly compared to Splunk and especially Elasticsearch for such queries.

Key-value stores for performance

Other possible solutions for achieving rapid search results can be to use key-value stores (or a variation: key-document stores). However, if wildcard searches are required the document must be searchable, which it is not in a key-value store. The reason for having the requirement of wildcard searches is because partial searches are important in querying leaked data to for instance find all emails ending with *@icloud.com*, or finding the relation between users using the same IP address. If wildcard searches for fields are not important, one of the fastest solutions would be to use a key-document store where multiple keys point to the same value containing one or more documents. In such cases, the key must be unique. This could potentially be a good solution but requires that the data is structured to a high degree. Keys could then be indexed, but that would result in a large amount of key-document stores as there are multiple searchable fields that must be indexed as keys. Based on the experimental tests, the

results of Elasticsearch are so rapid that it is unnecessary to investigate restructuring data to take advantage of the key in a key-document (key-value) store.

General Results

The experimental results provides evidence that MySQL and MongoDB can be viable solutions for rapid search results by using indexes for exact searches. However, if new data is frequently added, these systems are not recommended as indexes must be recreated for the new data. For leaked data, this is not the optimal solution, as the index creation is time-consuming and is not a process a database administrator should have to do every time a new breach is available.

Elasticsearch and Splunk automatically handle indexing of recently added data. The latter two also provide faster results for exact & wildcard in general. On the other hand, there are also cons with both Splunk and Elasticsearch. Due to the way the indexing works in these systems, the querying of certain data types can be time-consuming operations. For instance, a 30x performance increase can be obtained by using seconds instead of milliseconds as timestamps in Elasticsearch [63].

The Splunk Enterprise Trial version has proven to be a solid database system for querying large amounts of data but did not provide the fastest response times for the queries and type of data tested in this thesis. For network log files or data that contains a specific date or timestamp, Splunk provides the best search possibilities. It is possible to speed up certain queries in Splunk by using the TERM() operator on field searches for strings containing symbols such as dots. When using the TERM() operator the full term will be matched instead of tokenized matching separated by symbols.

After having finished evaluating the performance of all database systems, external sources recommended testing the Splunk API as it potentially can deliver faster query responses than the web interface. Testing this has been added to future work.

Chapter 6

Conclusion

This thesis introduces a comparison of rapid searches in leaked data on common hardware and our research concludes that Elasticsearch is the best performing solution.

This is not surprising as it is a storage engine purposely made for full-text searches and the results from experiments and testing also confirm its superiority. It is however no guarantee that this is the best solution in terms of performance and rapid response times for other types of data, since the database system depends on the purpose of the data.

Elasticsearch delivers the best performance when the user has limited hardware resources if the task is performing rapid text searches, and returns a response time of 1.58 seconds on 100 million records of leaked data.

Based on the results of our work, Splunk can be optimized to be approximately 2 to 3.5 times as fast compared to utilizing the original configuration, coming second to Elasticsearch with a response time of 4.27 seconds on 100 million records.

Rapid results can also be accomplished by using MySQL or MongoDB for exact searches. If the data to be queried is static and not frequently updated, MongoDB and MySQL will deliver the best performance after indexes have been (re)created. The differences in these systems are less than a few milliseconds when optimized and are therefore not considered to be distinguishable.

The results show that MySQL is the fastest database system for 10 million records or less, and reaches a turning point at approximately 50 million records where queries are taking significantly longer to finish.

By creating polynomial functions based on the experimental results it is possible to estimate

an approximate query response time for a given number of records.

Hypothesis 1: *MongoDB Percon can be optimized to provide better performance than Elasticsearch, MySQL Percona or Splunk.*

Our work confirms that MongoDB provides the best performance on exact queries when using indexes and optimized configuration among the database systems.

Hypothesis 2: *Proper optimization can result in a significant performance improvement.*

Research confirms that optimizing the configuration for each database system results in improved performance. All database systems were faster after optimization, and a significant increase was seen specifically in Splunk. Minor differences were observed in MongoDB before and after configuration optimization. Not surprisingly, the most significant performance increase was achieved when using indexes for MongoDB and MySQL.

Goal 1: *Determine which database system should be used to achieve the fastest possible search results on large amounts of leaked data.*

The research in this thesis concludes that Elasticsearch is the best solution for the fastest search results on leaked data, ultimately fulfilling this goal.

Goal 2: *Utilize optimization methods to achieve rapid search results when using inexpensive hardware that is typical for the average user.*

Based on the experiments and results from our work, several optimization methods can successfully be applied to achieve faster query response times on common hardware.

Goal 3: *Achieve rapid search results without the use of indexes in MongoDB and MySQL for exact- and wildcard searches.*

From research and testing, no optimization methods were discovered that resulted in a per-

formance increase to this extent. This goal was therefore not accomplished. Optimization techniques are available but did not provide a sufficient performance increase for results to be considered rapid.

Goal 4: *Achieve a query response time of fewer than five seconds for the maximum records tested.*

This goal was accomplished with the database systems Elasticsearch and Splunk. Querying maximum records with Splunk resulted in an average response time of 4.27 seconds. Elasticsearch was the most rapid system with an average of 1.58 seconds on maximum records, accomplishing this goal.

Almost all goals defined in the scope of this thesis were successfully accomplished and the hypotheses confirmed. The results of our work are not necessarily groundbreaking but serve as a guideline when selecting a database system for rapid search results in large sets of leaked data hosted on a single node. Our results also enable further research into testing new performance optimization techniques and comparisons with other database systems.

Chapter 7

Future Work

Data Types

1. Perform experiments with new data types.
2. Compare the differences in experimental results with current observations.

Hardware

1. Allocate 64 GB RAM and solid-state drives of 2TB or more to the pool of resources.
2. Perform experiments again using better hardware, and compare the results accordingly.
3. Outline a mathematical function for query response time with respect to hardware specifications and number of records.

Computational Distribution

1. Set up a cluster of 4-8 nodes in a distributed computational network.
2. Configure sharding for each node in the cluster.
3. Include Apache Hadoop in the list of database systems to be tested in a clustered environment.
4. Compare the performance of Hadoop against Elasticsearch or other clustered database systems, i.e Redis.

5. Present a new paper revealing the differences in results of utilizing a cluster versus a single node for read operations.

Utilize the Splunk API

1. Re-test Splunk's query response time using the API instead of web interface and compare response times.

References

[1] T. Hunt, *Have I Been Pwned: Check if your email has been compromised in a data breach*, [Accessed: March 6th 2019]. [Online]. Available: https://haveibeenpwned.com/.

[2] WeLeakInfo, *We Leak Info - Data Breach Search Engine*, [Accessed: May 20th 2019]. [Online]. Available: https://weleakinfo.com/.

[3] Leakedsource, *Find the source of your leaks*, [Accessed: May 20th 2019]. [Online]. Available: https://leakedsource.ru/.

[4] G. Dodig-Crnkovic, *Scientific Methods in Computer Science*, [Accessed: May 10th 2019], Dec. 2002. [Online]. Available: https://users.dcc.uchile.cl/~cgutierr/cursos/INV/crnkovic.pdf.

[5] K. D. Foote, *A Brief History of Database Management*, [Accessed: March 8th 2019], Mar. 2017. [Online]. Available: https://www.dataversity.net/brief-history-database-management/.

[6] CodeCademy, *What is a Relational Database Management System? — Codecademy*, [Accessed: March 20th 2019]. [Online]. Available: https://www.codecademy.com/articles/what-is-rdbms-sql.

[7] BrightPlanet, *Structured vs. Unstructured data*, [Accessed: April 29th 2019], Jun. 2012. [Online]. Available: https://brightplanet.com/2012/06/structured-vs-unstructured-data/.

[8] C. Taylor, *Structured vs. Unstructured Data*, [Accessed: March 19th 2019], Mar. 2018. [Online]. Available: https://www.datamation.com/big-data/structured-vs-unstructured-data.html.

[9] *STRUCTURED DATA IN A BIG DATA ENVIRONMENT*, [Accessed: March 15th 2019]. [Online]. Available: https://www.dummies.com/programming/big-data/engineering/structured-data-in-a-big-data-environment.

[10] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman, *UNDERSTANDING UNSTRUCTURED DATA*, [Accessed: March 15th 2019]. [Online]. Available: https://www.dummies.com/programming/big-data/understanding-unstructured-data.

[11] GeeksforGeeks, *Difference between Structured, Semi-structured and Unstructured data - GeeksforGeeks*, [Accessed: May 10th 2019]. [Online]. Available: https://www.geeksforgeeks.org/difference-between-structured-semi-structured-and-unstructured-data/.

[12] S. Yegulalp, *What is NoSQL? NoSQL databases explained — InfoWorld*, [Accessed: March 20th 2019], Dec. 2017. [Online]. Available: https://www.infoworld.com/article/3240644/what-is-nosql-nosql-databases-explained.html.

[13] M. Rouse, *What is relational database? - Definition from WhatIs.com*, [Acecssed: March 20th 2019], May 2018. [Online]. Available: https://searchdatamanagement.techtarget.com/definition/relational-database.

[14] Z. Tejada, M. Wilson, A. Buck, and M. Wasson, *Traditional relational database solutions*, [Accessed: March 20th 2019], Feb. 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/.

[15] V. Abramova, J. Bernardino, and P. Furtado, *Experimental Evaluation of NoSQL Databases*, [Accessed: May 11th 2019], Oct. 2014. [Online]. Available: https://www.researchgate.net/publication/307795516_Experimental_Evaluation_of_NoSQL_Databases.

[16] Pivotal, *Understanding NoSQL*, [Accessed: March 23rd 2019]. [Online]. Available: https://spring.io/understanding/NoSQL.

[17] DB-Engines, *Document Stores - DB-Engines Encyclopedia*, [Accessed: March 23rd 2019]. [Online]. Available: https://db-engines.com/en/article/Document+Stores.

[18] ——, *Wide Column Stores - DB-Engines Encyclopedia*, [Accessed: March 23rd 2019]. [Online]. Available: https://db-engines.com/en/article/Wide+Column+Stores.

[19] C. Alvarez, *NoSQL database: about quality attributes. Understanding first before choosing*, [Accessed: March 23rd 2019], Jan. 2017. [Online]. Available: http://www.tisa-software.com/news/blog/219-nosql-database-about-quality-attributes-understanding-first-before-choosing.

[20] DB-Engines, *Search Engines - DB-Engines Encyclopedia*, [Accessed: March 23rd 2019]. [Online]. Available: https://db-engines.com/en/article/Search+Engines.

[21] S. Choudhury, *A Busy Developer's Guide to Database Storage Engines - The Basics - The Distributed SQL Blog*, [Accessed: May 10th 2019], Jun. 2018. [Online]. Available: https://blog.yugabyte.com/a-busy-developers-guide-to-database-storage-engines-the-basics/.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, *The log-structured merge-tree (LSM-tree)*, [Accessed: May 20th 2019], Jun. 1996. DOI: 10.1007/s002360050048. [Online]. Available: https://www.researchgate.net/publication/226763355_The_log-structured_merge-tree_LSM-tree.

[23] C. D. Manning, P. Raghavan, and H. Schütze, *A first take at building an inverted index*, [Accessed: May 10th 2019], Apr. 2009. [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/a-first-take-at-building-an-inverted-index-1.html.

[24] M. Rouse, *What is database management system (DBMS)? - Definition from WhatIs.com*, [Accessed: March 23rd 2019], May 2019. [Online]. Available: https://searchsqlserver.techtarget.com/definition/database-management-system.

[25] Percona, *Percona Server for MySQL*, [Accessed: March 23rd 2019]. [Online]. Available: https://www.percona.com/software/mysql-database/percona-server.

[26] Snusbase, *Snusbase - Database Search Engine*, [Accessed: May 20th 2019]. [Online]. Available: https://snusbase.com/.

[27] Percona, *Percona Server for MongoDB*, [Accessed: March 23rd 2019]. [Online]. Available: https://www.percona.com/software/mongo-database/percona-server-for-mongodb.

[28] M. Makadia, *What Is Elasticsearch and How Can It Be Useful? - DZone Database*, [Accessed: March 23rd 2019], Oct. 2017. [Online]. Available: https://dzone.com/articles/what-is-elasticsearch-and-how-it-can-be-useful.

[29] S. Paul, *Database Systems Performance Evaluation Techniques*, [Accessed: May 11th 2019], Nov. 2008. [Online]. Available: https://www.cse.wustl.edu/~jain/cse567-08/ftp/db/index.html.

[30] S. Bennacer, *"Hot-Warm" Architecture in Elasticsearch 5.x*, [Accessed: May 11th 2019], Jan. 2017. [Online]. Available: https://www.elastic.co/blog/hot-warm-architecture-in-elasticsearch-5-x.

[31] tutorialspoint, *MongoDB - Sharding*, [Accessed: May 11th 2019]. [Online]. Available: https://www.tutorialspoint.com/mongodb/mongodb_sharding.htm.

[32] Philipp, *Difference between Sharding And Replication on MongoDB*, [Accessed: May 11th 2019], Sep. 2015. [Online]. Available: https://dba.stackexchange.com/questions/52632/difference-between-sharding-and-replication-on-mongodb/53705#53705.

[33] C. Kvalheim, *Sharding*, [Accessed: May 11th 2019], Oct. 2015. [Online]. Available: http://learnmongodbthehardway.com/schema/sharding/.

[34] A. Nikitin, *Transparent Hugepages: measuring the performance impact*, [Accessed: May 21st 2019], Aug. 2017. [Online]. Available: https://alexandrnikitin.github.io/blog/transparent-hugepages-measuring-the-performance-impact/.

[35] Splunk, *Transparent huge memory pages and Splunk performance*, [Accessed: May 21st 2019]. [Online]. Available: https://docs.splunk.com/Documentation/Splunk/6.1.3/ReleaseNotes/SplunkandTHP.

[36] V. Pandey, *XFS vs EXT4 – Comparing MongoDB Performance on AWS EC2*, [Accessed: May 11th 2019], Sep. 2017. [Online]. Available: https://scalegrid.io/blog/xfs-vs-ext4-comparing-mongodb-performance-on-aws-ec2/.

[37] S. Voultepsis, *Linux OS Tuning for MySQL Database Performance*, [Accessed: May 11th 2019], Jul. 2018. [Online]. Available: https://www.percona.com/blog/2018/07/03/linux-os-tuning-for-mysql-database-performance/.

[38] B. Cane, *Improving Linux System Performance with I/O Scheduler Tuning*, [Accessed: May 11th 2019], May 2017. [Online]. Available: https://blog.codeship.com/linux-io-scheduler-tuning/.

[39] A. Tonete, *Five Tips to Optimize MongoDB*, [Accessed: May 11th 2019], Mar. 2018. [Online]. Available: https://www.percona.com/blog/2018/03/22/five-tips-to-optimize-mongodb.

[40] S. Sanfilippo, *antirez/redis: Redis is an in-memory database that persists on disk. The data model is key-value, but many different kind of values are supported: Strings, Lists, Sets, Sorted Sets, Hashes, HyperLogLogs, Bitmaps.* [Accessed: May 20th 2019], May 2019. [Online]. Available: https://github.com/antirez/redis.

[41] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs cassandra", *ACM Digital Library*, Jul. 2013, [Accessed: May 11th 2019]. DOI: 10.1145/2494444.2494447. [Online]. Available: https://dl.acm.org/citation.cfm?id=2494447.

[42] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases", *IEEE Conference Publication*, Aug. 2013, [Accessed: May 11th 2019]. DOI: 10.1109/PACRIM.2013.6625441. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6625441.

[43] Z. Parker, S. Poe, and S. Vrbsky, *Comparing NoSQL MongoDB to an SQL DB*, [Accessed: May 11th 2019], Apr. 2013. DOI: 10.1145/2498328.2500047. [Online]. Available: https://www.researchgate.net/publication/261848669_Comparing_nosql_mongodb_to_an_sql_db.

[44] VMware, *ESXi — Bare Metal Hypervisor — VMware*, [Accessed: April 5th 2019]. [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html.

[45] N. Prebensen, *master_thesis_scripts*, [Accessed: May 23rd 2019], May 2019. [Online]. Available: https://github.com/nicolaipre/master_thesis_scripts.

[46] Percona, *Download Percona Server for MySQL 8.0*, [Accessed: May 23rd 2019]. [Online]. Available: https://www.percona.com/downloads/Percona-Server-LATEST/.

[47] A. Rubin, *MySQL 5.7 Performance Tuning Immediately After Installation*, [Accessed: April 29th 2019], Oct. 2016. [Online]. Available: https://www.percona.com/blog/2016/10/12/mysql-5-7-performance-tuning-immediately-after-installation/.

[48] V. Tkachenko, *MySQL 8 is not always faster than MySQL 5.7*, [Accessed: May 21st 2019], Feb. 2019. [Online]. Available: https://www.percona.com/blog/2019/02/21/mysql-8-is-not-always-faster-than-mysql-5-7/.

[49] Percona, *Download Percona Server for MongoDB 4.0*, [Accessed: May 23rd 2019]. [Online]. Available: https://www.percona.com/downloads/percona-server-mongodb-LATEST/.

[50] Elastic, *Install Elasticsearch with Debian Package — Elasticsearch Reference [7.1] — Elastic*, [Accessed: May 23rd 2019]. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/deb.html.

[51] Splunk, *Free Trials and Downloads — Splunk*, [Accessed: May 23rd 2019]. [Online]. Available: https://www.splunk.com/en_us/download.html.

[52] S. Documentation, *Optimize Splunk for peak performance*, [Accessed: May 12th 2019]. [Online]. Available: https://docs.splunk.com/Documentation/Splunk/7.2.6/Admin/OptimizeSplunkforpeakperformance.

[53] S. Delaney and J. Champagne, *The Jiffy Lube Quick Tune-up For Your Splunk Environment*, [Accessed: May 12th 2019], 2016. [Online]. Available: https://conf.splunk.com/files/2016/slides/jiffy-lube-quick-tune-up-for-your-splunk-environment.pdf.

[54] A. Nekkanti, S. Pal, and T. Anwar, *Harnessing Performance and Scalability with Parallelization*, [Accessed: May 12th 2019], 2016. [Online]. Available: https://conf.splunk.com/files/2016/slides/harnessing-performance-and-scalability-with-parallelization.pdf.

[55] S. Documentation, *Configure batch mode search*, [Accessed: May 12th 2019]. [Online]. Available: https://docs.splunk.com/Documentation/Splunk/7.2.6/Knowledge/Configurebatchmodesearch.

[56] Splunk, *Parallelization settings*, [Accessed: May 21st 2019]. [Online]. Available: https://docs.splunk.com/Documentation/Splunk/7.2.6/Capacity/Parallelization.

[57] D. Pilato, *Wildcard query returns null when Uppercase Letters are used*, [Accessed: May 12th 2019], Jul. 2015. [Online]. Available: https://discuss.elastic.co/t/wildcard-query-returns-null-when-uppercase-letters-are-used/24822.

[58] Splunk, *Indexes, indexers, and indexer clusters*, [Accessed: May 12th 2019]. [Online]. Available: https://docs.splunk.com/Documentation/Splunk/6.3.0/Indexer/Aboutindexesandindexers.

[59] PPape, *Why is IP address Searching/Matching so slow?*, [Accessed: May 12th 2019], Nov. 2016. [Online]. Available: https://answers.splunk.com/answers/476062/why-is-ip-address-searchingmatching-so-slow.html.

[60] J. Loisel, *ELASTICSEARCH: OPTIMIZATION GUIDE*, [Accessed: May 12th 2019], Sep. 2018. [Online]. Available: https://octoperf.com/blog/2018/09/21/optimizing-elasticsearch/.

[61] T. Eskildsen, *Memory is overrated*, [Accessed: May 12th 2019], Jun. 2013. [Online]. Available: https://sbdevel.wordpress.com/2013/06/06/memory-is-overrated/.

[62] N. Tanya, *MongoDB vs MySQL : Understanding the difference*, [Accessed: May 12th 2019], Apr. 2018. [Online]. Available: `https://blog.resellerclub.com/mongodb-vs-mysql-comparison/`.

[63] C. Price-Austin, *30x Faster Elasticsearch Queries*, [Accessed: May 12th 2019], Dec. 2016. [Online]. Available: `https://engineering.mixmax.com/blog/30x-faster-elasticsearch-queries`.

Appendices

Appendix A

Scripts

A.1 Python Data Generator

Listing A.1: Data generator script

```
1
2  import os
3  import sys
4  import uuid
5  import random
6  import string
7  import hashlib
8
9
10 from random import getrandbits
11 from ipaddress import IPv4Address, IPv6Address
12
13
14 if len(sys.argv) < 3:
15     sys.exit('Usage: python3 %s <amount> <output file>' % sys.argv[0])
16
17
18
19 # Define argument variables
20 amount   = sys.argv[1]
21 outfile  = sys.argv[2]
22
23
24 domains = ["hotmail.com", "hotmail.co.uk", "uia.no", "gmail.com", "aol.
       ↪ com", "mail.com" , "yahoo.com", "outlook.com", "icloud.com"]
25
26
```

```
27 def generateIp():
28     bits = getrandbits(32)
29     addr = IPv4Address(bits)
30     addr_str = str(addr)
31     return addr_str
32 
33 
34 def generateString(minLength=4, maxLength=20):
35     tmpLen = random.randint(minLength, maxLength)
36     lettersAndDigits = string.ascii_letters + string.digits
37     return ''.join(random.choice(lettersAndDigits) for i in
           range(tmpLen))
38 
39 
40 def generateEmail(username):
41     return username + '@' + random.choice(domains)
42 
43 
44 # Main
45 with open(outfile, "w+") as fp:
46     for i in range(int(amount)):
47         username = generateString()
48         password = generateString()
49         salt = uuid.uuid4().hex[:5]
50         ip = generateIp()
51         email = generateEmail(username)
52         password_hash = hashlib.sha512(password.encode('utf-8') + salt.
               ↪ encode('utf-8')).hexdigest()
53 
54         fp.write( str(username) + "," + str(email) + "," +
               str(password) + "," + str(password_hash) + "," +
               str(salt) + "," + str(ip) + "\n" )
```

A.2 Python Data Parser

Listing A.2: Python parser

```
1
2  import io
3  import os
4  import re
5  import sys
6  import json
7  import time
8  import pprint
9  import hashlib
10 import requests
11 from multiprocessing import Process
12
13 from elasticsearch import Elasticsearch
14 from elasticsearch import helpers
15
16 from modules import mysql
17 from modules import mongo
18 from modules import utils
19
20 if  len(sys.argv) < 7:
21     sys.exit('Usage: python3 %s <file> <leak-name> <chunk-size> <db-
       ↪ type> <host> <port>' % sys.argv[0])
22
23 if not os.path.exists(sys.argv[1]):
24     sys.exit('ERROR: file "%s" was not found!' % sys.argv[1])
25
26 # Define argument variables
27 filename   = sys.argv[1]
28 leakname   = sys.argv[2]
29 chunk_size = sys.argv[3]
30 db_type    = sys.argv[4]
31 host       = sys.argv[5]
32 port       = sys.argv[6]
33
34 # Other variables used along the road...
35 tempArray = []
36 arrayCount = 0
37 goodLines = 0
38 badLines = 0
39
40 dryRun = False
```

```
41 totalLines = utils.sumlines(filename)
42
43 # Output file to write bad lines to.
44 ignored = open(filename + "_ignored", "w")
45
46 # Ask for the regex:
47 regex = input("Input regex: ")
48
49 # Ask if we want to perform a dry run
50 run = input("Perform dry run? [y/n]: ")
51 if (run == "y"):
52     dryRun = True
53
54
55 # Function for writing data to database, for given database type
56 def dataWriter(data, db):
57
58     # MySQL settings
59     if db == 'mysql':
60
61         # Open connection for this chunk
62         con = mysql.connect("127.0.0.1", "3306", "schema", "root", "
               ↪ password")
63
64         fullQuery = """
65                 INSERT INTO breaches(
66                       `username`
67                     ,`email`
68                     ,`password`
69                     ,`password_hash`
70                     ,`salt`
71                     ,`ip`
72                     ,`leak`
73                 ) VALUES """
74         for line in data:
75             args = (line['username'], line['email'], line['password'],
                   ↪ line['password_hash'], line['salt'], line['ip'], line
                   ↪ ['leak'])
76             data = '("%s", "%s", "%s", "%s", "%s", "%s", "%s"),'
77             fullQuery += data % args
78
79         fullQuery = fullQuery[:-1] + ";"
80         mysql.query(con, fullQuery)
81
82         # Finally, close connection for this chunk.
```

```
83          mysql.close(con)
84
85
86
87      # MongoDB settings
88      if db == 'mongodb':
89          con = mongo.connect(host, port)
90          database = "databaseName"
91          collection = "breaches"
92          mongo.insertMany(data, con, database, collection)
93          # Finally, close mongo connection
94          mongo.close(con)
95
96
97      # Elasticsearch settings
98      if db == 'elasticsearch':
99          es = Elasticsearch([{'host': host, 'port': port}])
100         es = Elasticsearch(timeout=30, max_retries=10, retry_on_timeout
                ↪ =True)
101
102         def generator():
103             for i in data:
104                 entry = {}
105                 entry['_index']  = 'indexName'
106                 entry['_type']   = 'breaches'
107                 entry['_id']     = i['md5sum']
108                 del i['md5sum'] # Delete md5sum from source data
109                 entry['_source'] = i
110                 yield entry
111
112         for success, info in helpers.parallel_bulk(es, generator(),
                ↪ thread_count=2, chunk_size=1000):
113             pass
114
115
116     # Splunk settings
117     if db == 'splunk':
118         pass
119
120
121 #endef
122
123
124 # Start the insertion loop
125 with io.open(filename, encoding="utf-8") as fp:
```

```
126     for line in fp:
127         line = line.strip()
128 
129         match = re.search(regex, line)
130 
131         if match:
132             doc = match.groupdict()
133             doc['leak'] = leakname
134 
135             # Nasty oneliner for generating a md5 hash of our sorted
                ↪ doc values, used to detect duplicate entries later on
                ↪ .
136             doc['md5sum'] = hashlib.md5(json.dumps(doc, sort_keys=True)
                ↪ .encode('utf-8')).hexdigest()
137 
138             goodLines += 1
139             utils.progressbar((goodLines + badLines), totalLines)
140 
141 
142             # If dryrun is false, we are doing a run where we want to
                ↪ actually insert data:
143             if not dryRun:
144                 tempArray.append(doc)
145 
146                 # In this section, we check if the amount of documents
                    ↪ reach the specified chunk size, and then insert
                    ↪ the data if this is true.
147                 if (arrayCount < int(chunk_size) - 1):
148                     arrayCount += 1
149                 else:
150                     dataWriter(tempArray, db_type) # TODO?: create new
                        ↪ thread here, pass data to thread, repeat
                        ↪ process... outsource insertion job to threads
                        ↪ ..?
151                     del tempArray[:]
152                     arrayCount = 0
153 
154 
155         else: # if not match: #log bad lines to file
156             badLines += 1
157             ignored.write("%s\n" % line)
158 
159 
160 
161 
```

```
162 # Insert the rest | the reason this little chunck is here, is because
    ↪ if we dont reach our chunk size on the last loop, we will have
    ↪ data in memory that is not inserted to the database.
163 if not dryRun:
164     dataWriter(tempArray, db_type)
165
166
167
168 # Close file containing bad lines
169 ignored.close()
170
171
172 # Stats
173 print("\n")
174 print("--- Finished processing file ---\n")
175 print("File: {}".format(filename))
176 print("Leak: {}".format(leakname))
177 print("Regex: {}".format(regex))
178 print("")
179 print("Total lines: {}".format(totalLines))
180 print("Total good: {}".format(goodLines))
181 print("Total bad: {}".format(badLines))
182 print("\n--")
```

A.3 MySQL Configuration File

Listing A.3: MySQL configuration file

```
[mysqld]
pid-file     = /var/run/mysqld/mysqld.pid
socket       = /var/run/mysqld/mysqld.sock
datadir      = /mnt/extended/mysql
log-error    = /var/log/mysql/error.log


## Optimization below...

# files
innodb_file_per_table
innodb_log_files_in_group=8
innodb_open_files=65535

# buffers
innodb_buffer_pool_size=16G
innodb_buffer_pool_instances=8
innodb_log_file_size=1G
innodb_flush_method=O_DIRECT


# tune
innodb_doublewrite=0
innodb_thread_concurrency=0
innodb_flush_log_at_trx_commit=0
innodb_flush_method=O_DIRECT
innodb_max_dirty_pages_pct=90
innodb_max_dirty_pages_pct_lwm=10
innodb_lru_scan_depth=2048
innodb_page_cleaners=4
join_buffer_size=256K
sort_buffer_size=256K
innodb_use_native_aio=1
innodb_stats_persistent=1

# perf special
innodb_adaptive_flushing=1
innodb_read_io_threads=4
innodb_write_io_threads=4
innodb_io_capacity=1500
innodb_io_capacity_max=2500
```

```
42 innodb_purge_threads=4
43 innodb_adaptive_hash_index=0
```

A.4 MongoDB Configuration File

Listing A.4: MongoDB configuration file

```
storage:
  dbPath: /mnt/extended/mongodb
  journal:
    enabled: true
  engine: wiredTiger

  wiredTiger:
    engineConfig:
      cacheSizeGB: 16
      statisticsLogDelaySecs: 0
      journalCompressor: snappy
      directoryForIndexes: false
    collectionConfig:
      blockCompressor: snappy
    indexConfig:
      prefixCompression: true


systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

processManagement:
  fork: true
  pidFilePath: /var/run/mongod.pid

net:
  port: 27017
  bindIp: 0.0.0.0
  maxIncomingConnections: 65536
```

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT