Timo Myyrä

Improving Digital Asset Management Search in the Case Company

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master’s Thesis

29 March 2022

Abstract

Author:	Timo Myyrä
Title:	Improving Digital Asset Management Search in the Case Company
Number of Pages:	71 pages + 7 appendices
Date:	29 March 2022
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	
Supervisors:	Sami Sainio, Lecturer

This work was requested by a company modernizing their digital asset management system. The earlier version of the system was hard to maintain, slow and had many features which were no longer in use by clients. As the company could not find suitable existing system, they decided to develop a new system based on the old one. The new design would only keep features essential to the digital asset management and dropping others like support for product order handling that existed in the old system. After the first customer migrations into the new system, the general feedback was that the search was not functioning as expected. While the search was fast, the clients failed to find what they were looking for in some cases. The company requested for an investigation to be made to find out what problems current implementation has, what features are needed in the system and if the current architecture can provide them.

The research was done as a case study on the new system. To identify what search features are needed the system was examined from different view points. The work used five view points, interview done with project management, customer survey from users, log analysis to look at past behavior, examining Google search and a digital asset management market review. Based on the findings from these view points a requirement specification was created.

With the requirements defined the last part of the study was dedicated to analyze whether the current architecture can provide all the identified features. These tests were done with the PostgreSQL database using various search methods. The test results indicate that the PostgreSQL can be used to implement all the required features. The study found several places were the search may be improved and identified many paths for future work.

Keywords: digital asset management, information retrieval, search, postgresql

Tiivistelmä

Tekijä:	Timo Myyrä
Otsikko:	Digitaalisen aineistohallinnan haun parantaminen tapausyrityksessä
Sivumäärä:	71 sivua + 7 liitettä
Aika:	29.3.2022
Tutkinto:	Insinööri (YAMK)
Tutkinto-ohjelma:	Tieto–ja viestintätekniikka
Ammatillinen pääaine:	
Ohjaajat:	Lehtori Sami Sainio

Tämä työ suoritettiin digitaalista aineistohallintaa uudistavan yrityksen pyynnöstä. Järjestelmän aiempi versio oli työläs ylläpitää, hidas ja sisälsi paljon ominaisuuksia, joita asiakkaat eivät enää käyttäneet. Koska yritys ei löytänyt sopivaa korvaava aineistohallintaa, päättivät he kehittää uuden vanhan pohjalta. Uusi järjestelmä säilytti vain aineistohallinnalle olennaiset ominaisuudet tiputtaen muut, kuten tilausten käsittelyn. Ensimmäisten asiakasmigraatioiden jälkeen, yleinen palaute järjestelmästä oli, että haku ei toiminut odotetusti. Vaikka haku oli nopea, se ei palauttanut odotettuja osumia kaikissa tilanteissa. Yritys pyysi tutkimuksen tekemistä haun ongelmien ja hakutoimintojen tarpeen selvittämiseen, sekä kartoittamaan, onko nykyinen arkitehtuuri riittävä tarjoamaan ne.

Tutkimus suoritettiin tapaustutkimuksena uudesta järjestelmästä. Tarpeellisten hakuominaisuuksien tunnistamista varten uutta järjestelmää tarkasteltiin eri näkökulmista. Työssä käytettiin viittä näkökulmaa, projektijohdon haastatteluja, käyttäjille tehtyä asiakaskyselyä, käyttäjien käyttäytymistä vanhojen lokitietojen pohjalta, Google-hakuun perehtymistä ja digitaalisen aineistohallinnan markkinakatsausta. Näiden näkökulmien löydösten perusteella luotiin vaatimusmäärittely.

Vaatimusmäärittelyn perusteella analysoitiin, kykeneekö uusi arkitehtuuri tarjoamaan kaikki vaaditut ominaisuudet. Nämä testit suoritettiin PostgreSQL-tietokannassa suoritetuilla erilaisilla hakumetodeilla. Testitulokset osoittivat, että PostgreSQL kykenee tarjomaan vaaditut ominaisuudet. Tämä työ tunnisti useita kohtia haun parantamiselle ja mahdollisia jatkotutkimuksen paikkoja.

Avainsanat: digitaalinen aineistohallinta, tiedonhaku, haku, postgresql

Licenses

I decided to publish my thesis work under the Creative Commons Attribution-ShareAlike 4.0 International License because I strongly believe that you as reader deserve the freedom to copy, share and modify this work and if you do modify it, it is fair to give these same permissions to the others.

Acknowledgement

Thanks to Panu Leppäniemi, Patrik Luoto and Patrick Ausderau for the LaTeX thesis template. This allowed me to finally use and learn LaTeXtypesetting. Besides teaching typesetting the research extended my database knowledge and gave a glimpse on vast the information retrieval field is. Also, big thanks to my instructor Sami for providing invaluable suggestions and ideas throughout the writing process.

And finally I would like to extend a big thank you to my wife and kids for having patience and listening my endless blabbering about this thesis.

Vantaa, 29 March 2022
Timo Myyrä

Contents

List of Abbreviations

ACID:	Atomicity, Consistency, Integrity, Durability.
ACL:	Access Control List.
AI:	Artificial Intelligence.
API:	Application Programming Interface.
B2B:	Business-to-Business.
CBIR:	Content-Based Image Retrieval.
CLIR:	Cross-Language Information Retrieval.
CMS:	Content Management System.
CV:	Computer Vision.
DAM:	Digital Asset Management.
DCMI:	Dublin Core Metadata Initiative.
DMS:	Document Management System.
DRM:	Digital Rights Management.
FTS:	Full Text Search.
GIN:	General Inverted Index.
GiST:	Generalized Search Tree.
HTTP:	Hypertext Transfer Protocol.
IR:	Information Retrieval.
JSON:	Javascript Object Notation.
ML:	Machine Learning.
NLP:	Natural Language Processing.
OCR:	Optical Character Recognition.
PAM:	Production Asset Management.
PDF:	Portable Document Format.
POS:	Part-of-speech.
RBAC:	Role-Based Access Control.

RDBMS:	Relational Database Management System.
REST-API:	Application Programming Interface using Representational State Transfer.
SaaS:	Software as a Service.
SQL:	Structured Query Language.
UI:	User Interface.
URL:	Universal Resource Locator.
UUID:	Universally Unique Identifier.
WCM:	Web Content Management.
XML:	Extensible Markup Language.

1 Introduction

This research was done at the behest of a company providing Business-to-Business (B2B) services who were in the middle of renewing their digital asset management system as the old system was not keeping up with the competition. This monolithic system combined features from online ordering and asset management making it technically complex and costly to develop and maintain. After reviewing the state of the old Digital Asset Management (DAM) it was seen that it would not be cost-effective to develop it further, so it was decided that it would be replaced entirely with new systems, each focusing on single specific area. This research covers the new system focusing on DAM.

The company initially sought to license an existing system as a base for this new DAM system but could not find a suitable candidate. The old system included many customer-specific extensions which were mandatory to have in the new system as well. To fulfill these requirements would have required extensive customization done in available DAM offerings, increasing their cost. The cheapest evaluated DAM providers were not providing significantly better features than the old system, so choosing them would not have given any edge in the market compared to the old system. The international DAM offerings would have provided most of the required features and would have added plenty of other useful ones on top of those. Problem with them was their licensing policy. To keep the existing customers it would have meant that the new system would have required multiple expensive licenses and in addition the instances would have still required costly customizations. Because of these the company decided that the replacement system would be developed from scratch. The task was given to subcontractor handling the maintenance of the existing system as they had already the insight and knowledge from the previous system.

The design goal for the new system was to produce modern application following current best practices[1]. These would include leveraging containerization, so the

[1] https://12factor.net/

application would be running in its own image container which would keep the environment of the application similar wherever it was deployed. By use of containers the application would be designed using microservices architecture where key functionalities are split into separate containers which work together instead of making a monolithic application. On the code level it would use functional programming techniques such as static typing to be able to catch programming errors during the compile-time. Automation would be used in the project to provide continuous integration and continuous development. This would mean the development team would make smaller deployments frequently which would be build and tested and deployed to production faster than was possible with the old system. The benefit of having an in-house developed DAM system is that in addition to providing it as a Software as a Service (SaaS) it can be licensed to customers internal use as well, as well as controlling the development the company can prioritize which direction the development would focus.

The new digital asset management system was marketed by its fast operations and familiar “Google-like” search. After first customer migrations were done the feedback from users was generally positive regarding the search as it was fast but on some cases the system did not return expected results. After altering the search, so it worked correctly on problematic queries, it started to exhibit problems in other search queries. The project management was not satisfied with current search as it is one of the key features of the DAM.

This research was done as a case study on the new DAM system aimed to answer three questions; what are the issues in the current search, what features should it provide and finally, is the current architecture capable in delivering those.

To find out how the search in the new DAM system should be implemented, the current search implementation was analyzed on how it works and what features it provides. This is then compared on what the project management want from the system and what the users of the system need in their work. Finally, the results are compared on what the current market leaders in DAM are providing for their customers. Analyzing results from these will allow creation of a requirement specification for improving the system if needed.

This thesis has been divided into eight chapters. The first chapter is intended to give background information on why the research was done. The next two chapters present the used research methods and give the necessary background theory related to search in DAM systems. The fourth chapter describes the current system details and presents the different view points on how to decide what features would be needed in the DAM. In the fifth chapter the view points are analyzed, so an accurate requirement specification can be defined which is then used as the basis of further system analysis done in the sixth chapter. The analysis presents more in-depth view on how the current system may be used to deliver the identified features in requirement specification. The last chapters present recommendations and discussion on findings followed by concluding summary.

2 Material and Methods

The research done as part of this thesis aims to identify problems in current search, what features it should provide and is the current DAM architecture capable on delivering them. The focus of the research is to try to identify what features are essential to provide sufficient search experience for customers. Other focus point is to analyze if the existing architecture and tooling is enough for the current and near-future needs. A case study was chosen as a method to research this.

The beginning of this study consists of literary review to gain theoretical background and doing current state analysis on the new system. The current state analysis is done to understand the new DAM system from different view points. The first view point is the technical view point, what the system is currently. Looking at the project documentation and interacting with the system to map out what kind of architecture the system is using, what tools are used in it, how it is working. The next view point is from project management by conducting in-depth interviews on how they view the current search should function. The interviews were done with qualitative approach to get the overview on what the project management viewed as important search features to have in the system. The interview also included questions covering other areas beyond the search features to get idea about the scope of the project and get estimates on how much resources would be available for implementing any search changes and to identify other potential limitations. Following the interviews, a customer survey was done to gain system users point of view, what search features they need in their work. The survey questions were based on the interview questions, but they were changed to only cover search features. The survey was done using quantitative approach. The users were asked to rate the overall system search in the current system and to provide ratings on various search features to get their view on what they saw as useful features. The last view point used in the study is from market leaders. They have long history on the market making their analysis helpful in laying road map on what direction the DAM should be heading. As all the market leaders provide proprietary systems the information comes from their documentation and marketing material making it biased but big themes what kind of search features are provided by them can be identified from them. Analyzing the findings of these different view points allows a requirement specification to be made. The specification is then used to do empirical analysis to test, if the current architecture and tooling is enough to cover those requirements with the hypothesis, that the current architecture and tooling is enough to handle current and near-future customer search needs.

3 Theoretical Background

This chapter is presenting the theoretical background needed to understand the DAM and technologies related to their working. The chapter starts with short introduction of Machine Learning (ML) as it is powering many Artificial Intelligence (AI) features included in DAM products. Many of these features are based on Natural Language Processing (NLP) and use of language models which are described next. This work focuses on searching so large section of theory is dedicated for Information Retrieval (IR) to define what is information and how to search it. The chapter closes on describing metadata and how DAM systems bind all the above together.

3.1 Machine Learning

ML is subfield of AI that “studies the ability to improve performance based on experience.” [1, p. 1] ML most useful in cases where there is plenty of example data available and the problem to be solved is hard or impossible to express formally. For example, it is very hard to express, if an image contains a cat. ML can accomplish this by sampling through huge amount of labeled images figuring out what makes cats stick out from images.

ML works by building *a model* from *data* it has available. The model is used to solve the task and task outcome may be used to further improve the existing model. [1, p. 651] The model dictates how ML system interprets data it receives. To create a model it must be *trained* to work in given problem. Training of a model requires data. The data should contain enough varied examples of things the model should be able to recognize. The quality of data affects much on the result of the model. [2, p. 42] There needs to be enough data for the model to learn from but with even small collection of data can be used to start the process. ML methods have become popular recently as increase in amount of available data and processing power made the ML algorithms suitable for many tasks, allowing them to outpace previous expert systems. [2, p. 10]

There are three main types of learning for ML based on the feedback the model is given, *supervised*, *unsupervised* and *reinforcement learning*. [1, p. 653] The most common learning method in ML is *the supervised learning*. In it the ML algorithm is given *labeled data* with aim that the algorithm can figure out an approximation for a function producing the labels for the data. It is usually used for either in regressions as shown in Figure 1 or in classification tasks as in Figure 2. In regression the model is used to predict numeric values based on the input data and in classification it tries to learn to label given inputs.

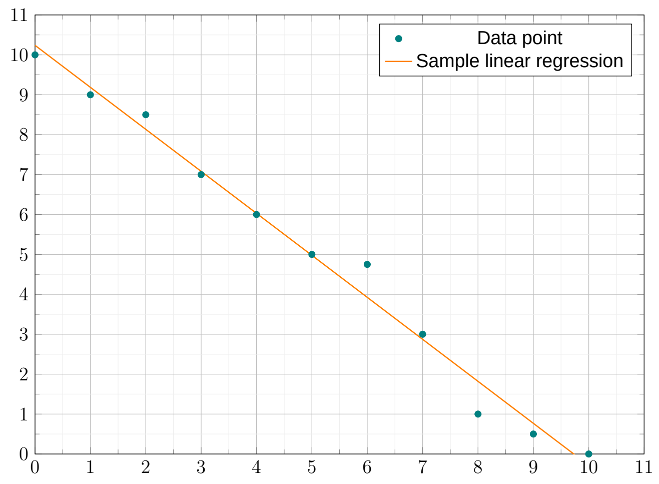


Figure 1: Example of Linear Regression.

The *unsupervised learning* is learning without giving the model any external feedback. The system is given unlabeled training data and the algorithm attempts to gain insights from it. The most common task for unsupervised learning is clustering [1, p. 653].

The third learning model is called the *reinforcement learning* in which the model is given feedback on the final outcome of its actions, either a positive (reward) or negative (punishment). Based on this feedback the model is left to teach itself how to maximize the rewards. [1, p. 789] This has been used with good results in game AI like with AlphaZero [3]. The AlphaZero can master games by being given only the basic rules of the game and then left to train through self-play.

Artificial Neural Networks, also known as *Deep learning* are next step from ML. They try to mimic how human brain functions, meaning many simple

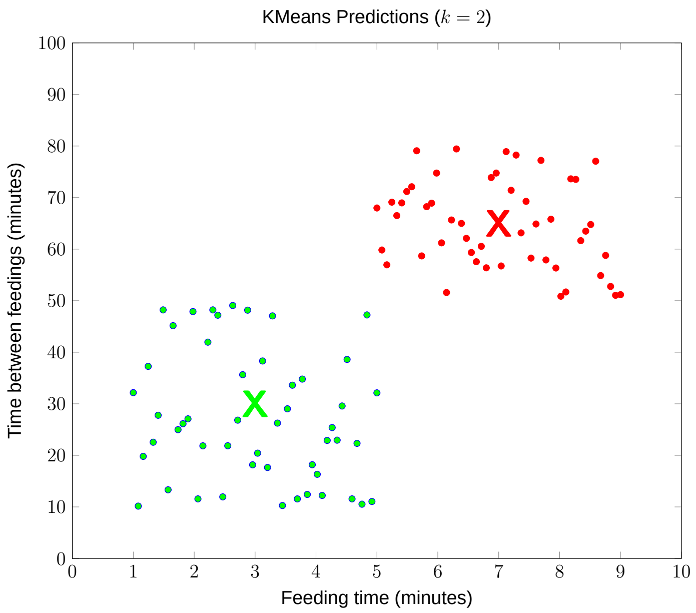


Figure 2: Example of Clustering of Results.

interconnected processors or nodes which work with input data through layers as shown in example network in Figure 3.

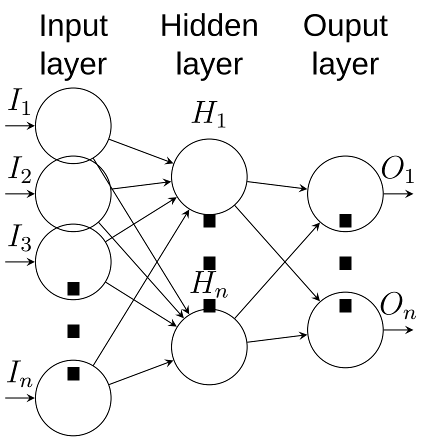


Figure 3: Example of Neural Network.

The nodes in the network have weights and threshold value. Once the input exceeds the node threshold, it is passed on the next layer on the network. [4] Where regular ML algorithms are verifiable by human how their predictions came to be, the deep networks might be so large that its impossible for human to work out exactly how the network came to its conclusion. Other notable difference is amount of time and processing power required for training. Where ML algorithm training can take several hours, typical training period for deep learning network might take a week. Even as it takes a lot longer to train, neural networks allow the algorithms to scale beyond what regular machine learning algorithms can provide. [2, p. 11]

3.2 Natural Language Processing

NLP is “a subfield of AI that refers to computational approaches to process, understand, and generate human language.” [5, Ch. 1] One common task for NLP is to parse documents. This process starts by parsing input data in useable form using a parser. These steps may include *speech-to-text* processing to convert spoken language into text. Once data is text-based it is *tokenized* e.g. split into sequence of words, and they may be further tagged based by their Part-of-speech (POS) as verbs, nouns etc. to resolve some ambiguities. [6] Depending on task the tokens are usually *normalized*. Normalization is a process to reduce the superficial differences in tokens by use of synonym matching, unifying term casing and use of diacritics, removing stop words and similar methods. [7, p. 28]. *Synonym matching* attempts to match token with its synonym to reduce the amount distinct tokens. The *stop words* are common and/or irrelevant words in the given language which will not be needed for accurate search. For English, the list of stop words might include “the”, “a”, “an” etc.

NLP is vast field, the IR described in upcoming Section 3.4 is a subset of NLP. Other common tasks where NLP is used include *text classification*, *information extraction*, *topic modeling* and *machine translation*. [8, Ch. 1] Text classification uses NLP to categorize given text based on the content, example use case would be email spam filtering. Other use for it could be to classify and thus organize massive amounts of data automatically. [8, Ch. 4] In information extraction text is parsed in attempt to identify and extract important segments of given problem domain out of text. The goal is to gather specific bits of information or *ontologies*. [9, p. 448] Ontologies describe objects and their features. [10, p. 2] These are stored in knowledge bases. Other use case would be to automatically apply tags to documents based on extracted parts of text [8, Ch. 5] Topic modeling is common method of NLP which is extensively used in document clustering and organizing large collections of text data. [8, Ch. 7]

The traditional heuristic NLP methods are still important building blocks, but the rise of computational resources has shifted the modern NLP to rely on ML as was presented Section 3.1. [5, Ch. 1]

3.2.1 Language Models

For NLP to help in above tasks, it needs to understand what it is working with. Natural languages are vague, ambiguous or cannot be formally defined which makes working with them hard. One option is to approximate the language with probabilistic *language model*. These models are one of the main tools used in NLP. These models may be used for many tasks. As they allow predicting what words are most likely to follow when given some input text, they can be used for text completion. Another common use is grammatical and spelling correction which can be implemented by calculating probabilities of word alternatives and pick the most likely. [1, p. 824] If there are models for multiple languages, these may be used to calculate most probable translation given text.

Simplest model represents language as independent words hence the name *bag-of-words model*. These work by examining each word individually without any extra context. It is a crude method, but it yields good results in tasks such as classification. [1, p.824]

An n-gram models extend the model to include context in it. Each word depends on the previous n words in n-gram model. [11, p.50] Simply put the n-gram tells how many units of history to consider when calculating probabilities. The first few n-grams are special cased and name *unigram* for 1-gram, *bigram* for 2-gram and *trigram* for 3-gram model. [1, p.824] N-gram models provide good results on *sentiment analysis*, *spam detection*, *author attribution*. An n-gram model working on n characters copes well with unknown words and compound words. It is particularly good at *language identification* and can achieve over 99% accuracy for short sentences.

3.2.2 Stemming and Lemmatization

Stemming is process of identifying *word stem* of given word and algorithms using stemming are called *stemmers*. There exists various methods how stemming can be achieved, the program may strip inflected suffices from words by using language-specific rules. The process works on most words, but irregular words will return wrong results. Other alternative is to have dictionary of words where stemming is just a look-up. The process is fast, but it will not work for unknown words. As stemming reduces many words to same stem it increases recall of the search while harming precision. [7, p. 34] Each language have different rules of stemming, although there are common algorithms which provide results which are close enough in practice for many languages.

The word stem returned by a stemmer might not be a valid *root word* in itself. For example using Snowball stemmer for words “run ran runs running runner” results in “run ran run run runner”, or by removing duplicates “run ran runner”. Another example where the stemmed version does not return a root word at all can be seen with words “categories category” which yields “categori categori”.

Stemming algorithms have two main categories for errors. The algorithm might incorrectly reduce several words from different contexts into same word stem, this is called *overstemming* or a false positive. The second error is the opposite, multiple words which are related to same concept are not stemmed into same word stem. This is called *understemming* or a false negative. [11, p. 84]

There exists various stemming algorithms or stemmers, here are few listed: [11, p. 83]

Porter stem Popular algorithm which is simple and fast. Downside is that only supports English language.

KStem Similar to Porter but less aggressive and faster

Snowball stemmer Vastly improved version of the Porter stemmer by the same author, adds support for other languages in addition to English.

Hunspell A combination of dictionary and rule-based stemmer supporting many languages.

Sometimes algorithmic stemming is not enough to get good results and further analysis of the text is needed. The *Lemmatization* is the process of finding the lemma for a given word by doing full morphological analysis for the text. [7, p. 33] Algorithms implementing it are called *lemmatizers* [7, p. 34]. This kind of analysis is more time-consuming than stemming but usually give more accurate results.

The benefit of using a stemmer or a lemmatizer varies by language. For English language their usage brings quite modest benefit but for languages with more complex morphology such as Finnish the results are often better. [7, p. 46]

3.3 Data Retrieval

Data and Information retrievals are search methods. The distinction between them is vague but Table 1 replicated from Information Retrieval [12, Ch. 1] presents how methods on how to tell them apart.

Table 1: Data Retrieval or Information Retrieval?

	Data Retrieval	Information Retrieval
Matching	Exact match	Partial match, best match
Inference	Deduction	Induction
Model	Deterministic	Probabilistic
Classification	Monothetic	Polythetic
Query language	Artifial	Natural
Query specification	Complete	Incomplete
Items wanted	Matching	Relevant
Error response	Sensitive	Insensitive

The data retrieval can be thought as fetching data the user already has idea where it exists in the system. This means the user may use artificial query language to express which records to fetch. Next section covers briefly methods of data retrieval before proceeding describing information retrieval in Section 3.4.

3.3.1 Data Retrieval Methods

The methods used in data retrieval work by matches. Simplest methods used are *equality tests*, does the data match exactly the given search query. Problem with this kind of search is that the user must give the search term exactly as it is given in the data for it to yield a match. This is hard to use when there are multiple fields to search and the fields content become larger or user does not recall exactly what to search for. The next step up from equality test is to expand the search with *pattern matching* like the wildcard or truncate matching. Wildcard matching can specify patterns such as string starts with given prefix, ends with given suffix or contains given substring within it.

To express more complex pattern a *regular expressions* can be used. Regular expressions are textual representation of pattern which is matched against string. It has rules such as “match text which begins with letter a and end in letter c”. This can be represented as regular expression “^a.*c$”. The “^” and “$” characters are anchors that anchor the matching to either start of search space or end of it. The “.” character has meaning of match any character and “*” has meaning preceding pattern needs to match zero or more times.

To distinction between data and information retrieval blurs a bit when using techniques such as *fuzzy matching* where instead of given strict match/not match the algorithm gives degree of matching. The *edit distance* [13, p. 211] and n-gram algorithms provide good support in to handling typing errors. [11, p. 80] The edit distance algorithms compare given input strings by how many single character operations are required for the inputs to match each other. For example the *Levenshtein distance* algorithm calculates the metric by allowing insertion, deletion and changing of a single character. [13, p. 213] N-gram algorithms break the strings to be matched into sets of N characters and compare how many of these sets are shared between the input strings.

Sometimes users know what they want but are unsure how write it down correctly, for example names of people and places. In these cases *phonetic matching* might provide best result. They provide fuzzy matching based on the pronunciation of the words. Most well known of these kinds of algorithms is the *Soundex* [7, p. 63]. The algorithm calculates four character length hash code for input string based on its English pronunciation. Then different inputs can be compared for similarity by looking how many Soundex code characters are shared between the inputs. The method is simple and well understood making many other phonetic algorithms based on the idea of Soundex but improving it to work on other languages or cover more characters.

3.4 Information Retrieval

Earlier Section 3.3 covered data retrieval where user uses exact search queries to fetch data. The difficulty with such methods is that the user is required to know what to search for to succeed. IR is different from the data retrieval in that its probabilistic process making use of linguistic knowledge rather than a deterministic one. It uses approximate or fuzzy matching and results are ordered by how relevant they are for given user query.

The goal of information retrieval system is to give its user the relevant information which is implemented by three main abstractions: [14, p. 2]

- Presentation of documents
- Presentation of user information need
- Comparing the above two presentations for matches.

To present documents the system needs to have them prepared and stored within its indexes. The second part is the user requesting some information from the system by formulating a query. The query is matched against the indices of the IR system to look for matches. The results are presented to the user, who then either accepts the result or gives feedback about incorrect results in a form new query. This gives the high-level model of the IR process which is presented in the Figure 4, the following sections cover each of the above steps in more detail.

The IR system works by units of document. A document might be anything the IR

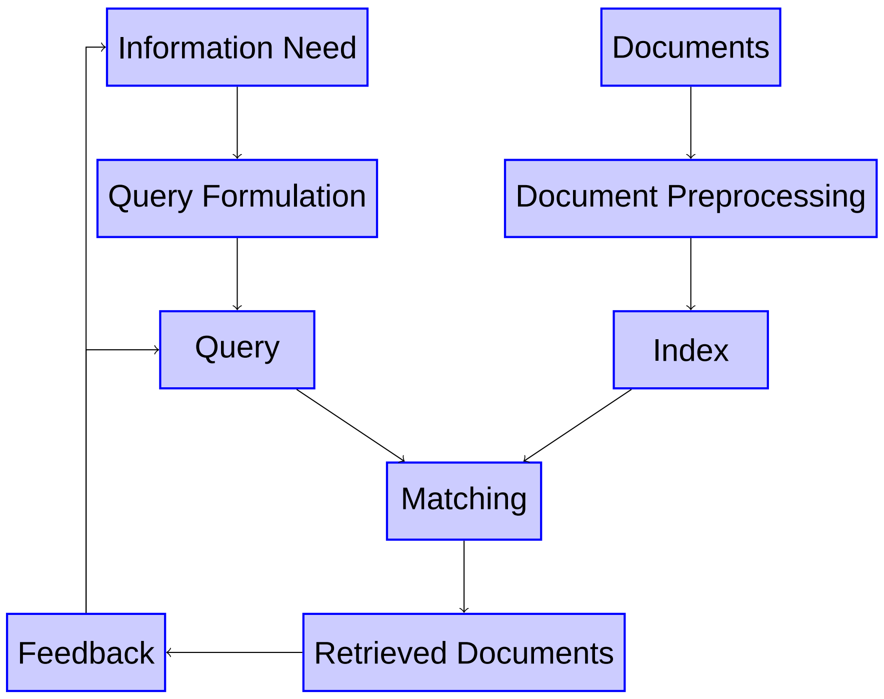


Figure 4: The Overview of the Information Retrieval Process.

system is built to handle, but this thesis covers digital asset management systems so makes assumption that document is a digital file or part of it. A group of documents is called a *collection* or a *corpus*. [7, p. 4]

Document consists of data which is usually either *structured* or *unstructured*. Structured data is based on some schema [15, p. 14]. The schema defines how the data is structured, so it is easy for machines to parse and handle as appropriate. Good example of structured data is simple Extensible Markup Language (XML) document with sample given in Listing 1.

```
<note>
  <to>Instructor</to>
  <from>Me</from>
  <heading>Reminder</heading>
  <body>Do not forget to return the thesis!</body>
</note>
```

Listing 1: Sample XML Document Contents.

Unstructured data is raw data, like plain text file which the machine can not parse without external aid. IR system do not require structured data to be able to work, but it can provide additional context information which can be helpful to have.

3.4.1 Document Preprocessing

The document undergoes a various preprocessing steps before it or part of it gets stored in the system storage. If the document is a file in file system, the IR system needs to parse the contents of the file based on the file type and the text content is further processed. Preprocessing parses documents using NLP [9, p. 446] which was described in Section 3.2. During the preprocessing for IR the parser may replace term with a related term or synonym which helps to reduce the number of indexed terms and thus improve precision. [7, p. 29] Other common task is removing stop words, as it generally improves the precision of the search. [7, p. 27] Diacritics may also be removed from the terms so that “café” and “cafe” will match each other. [7, p. 29]

Once the document has gone through preprocessing steps the results are indexed in the IR system. The most important indexing method is the *inverted index* [16, p. 16]. The inverted index is a kin to index which can be found at the end of the book. It stores each unique term and maps which document contains the term. In addition, the index might contain extra information such as weights and term frequencies [11, p. 47] Indexing in IR systems prepares the document so it can be later retrieved efficiently. There is two common indexing methods in use [14, p. 17]. Simplest indexing method is *binary indexing* were each word is either associated to the document or not. Other method called *weighted indexing* adds numerical weight value indicating how significant the word is for the given document. More information about weighting will be given in Section 3.4.2.

3.4.2 Information Retrieval Models

How the IR system derives the ranked list of potentially relevant documents from the given query is called the *IR model* [15, p. 15]. The model describes how the query and documents are represented, matched and how the query results are ranked. The traditional models are the Boolean, vector space and probabilistic models. [17, p. 11] Though each individual IR system use their own model, they are usually derived from the traditional models detailed below. [17, p. 11]

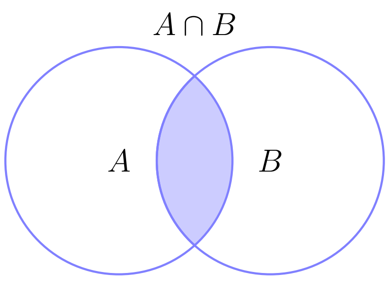


Figure 5: Example Venn Diagram of Boolean Union.

In the *Boolean model* documents are represented as a set of terms and matching is done using logical Boolean AND, OR, NOT operators with sample given in Figure 5. [11, p. 48] Downside of the model is that the AND and OR operators are totalitarian. The AND operator between multiple search terms is too restrictive as all terms must exist in the document for it to match. If 3 out of 4 terms match will not yield a match in this model. Similar but opposite problem is in OR operator as it will include all documents where any of the terms exists leading to information overload. Some models provide more relaxed AND and OR operators that they yield a match when either of the terms matches [18, p. 100]. For example “foo AND bar” would give match for documents with foo but not having bar in them.

Primary problem with Boolean model is that results are not easily ranked by relevance. The second traditional IR model called *vector space model* attempts to improve this by using vectors and matrices. In Figure 6. In vector space model documents and the search query are mapped into N-dimensional vector space as shown in Figure 6. The model returns the matches which are similar enough and ranks them according to the angle between them and the query vector. [11, p. 49]

Usually each term in a document is represented by single vector in the vector space. The ranking of vectors may be done using the *cosine similarity* [7, p. 158] which ranks based on how closely they point into same direction. The Equation 1 presents cosine similarity between vectors $\vec{a}$ and $\vec{b}$.

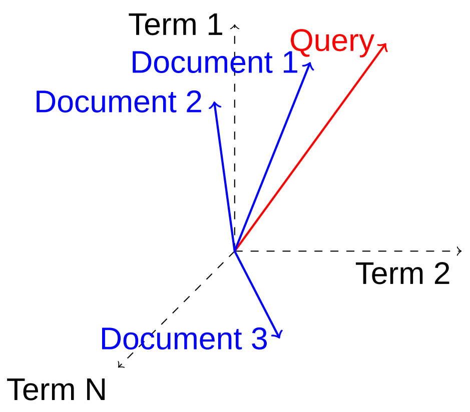


Figure 6: Example of Vector Space Model.

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}|\ \vec{b}} \tag{1}$$

The last of traditional models is called *probabilistic model* which uses probabilities to calculate what is the probability that given document is relevant to given search term. [19, p. 5] The most basic method to achieve this is to use *Bayes' Theorem* presented in Equation 2 to calculate the probability that document D is relevant for search term T: [15, p. 18]

$$P(D \mid T) = \frac{P(T \mid D)\, P(D)}{P(T)} \tag{2}$$

These are only the traditional models of IR. There also exists hybrid models such as *extended boolean model* [7, p. 14] which expands the standard boolean model with capabilities from vector space model allowing it to provide proximity matching and phrase searches.

Weights

Ranking of search results depends on the model used and the terms in the document. Boolean model treats all matched terms equally. To have some degree of relevancy ranking in the results the model may segment each document into smaller parts called *zones* [7, p. 110]. Usually document title, abstract and body are treated as different zones. Each zone may use its own index and by using several zones allows the IR system to do *weighted zone scoring*. [7, p. 110] The scoring allows boolean query to rank matching documents based on which zone the match was found for which the method also called *ranked boolean retrieval* [7, p. 112]. These weighs may be given by the user or automatically learned from examples. This is called *machine-learned relevance* [7, p.113]. The basics of the ML were given in Section 3.1.

The above only considers the existence of a search term, but typically it is beneficial to consider the frequency of a term as well. There are few common equations when dealing with term frequencies and most basic one is simple count of term occurrences in a corpus:

$$F_i = \sum_{j=1}^{N} f_{i,j} \tag{3}$$

The Equation 3 defines the term frequency F_i of term i as sum of occurrences of term i in each document in a corpus of N documents. [18, p. 101] This equation serves as good base, but it has problem that term counts do not directly relate to increase in relevancy, document with 1000 terms over one with 100 is not necessarily ten times more relevant.

$$TF_i = 1 + \log \sum_{j=1}^{N} f_{i,j} \tag{4}$$

Equation 4 uses logarithm to smooth out the score. This effectively changes the document into a *bag of words* and does ranking based solely on how many terms match the search query. [7, p. 117] Problem with above matching is that treats each term equally, but some terms are not really important for relevance such as stop words and other frequent terms within the corpus. If the all documents contain the term “medicine” it does not really add value as query term. To cope with these a helpful metric is *document frequency* df_i, a number of documents which contain term i [7, p. 118] which may be used to give more weight to search terms which occur less frequently in the corpus. This is called the *inverse document frequency* IDF_i:

$$IDF_i = \log \frac{N}{df_i} \tag{5}$$

Equation 5 shows inverse document frequency IDF_i where N is the number of documents in the corpus and df_i number of documents containing term i. The idea in inverse document frequency is that terms which occur on fewer documents would give more relevant results. [20, p. 5] The term frequency and inverse document frequency are commonly combined and used as weights in vector space models. [18, p. 101]

$$TFIDF_{i,j} = TF_{i,j} * IDF_i \tag{6}$$

3.4.3 Query Expansion

Challenge in the information retrieval is that queries given by users leave room for interpretation. This makes the information retrieval system unsure what the user has meant by the given query, situations where search term exits in two sets of documents with vastly different meanings. For example when given search term “bank”, does the user want to have information on river banks and financial institutions. Users of IR system can remedy this by refining their queries manually but the system itself can help to find the correct results by method of *query expansion* [14, p. 28]. There exists either global or local methods for achieve this. [7, p. 214] In global methods the system may reformulate or expand the query terms automatically without knowing the result. The local methods use iterative method with access to the result set and may include user in helping to hone on the target documents. The basic local methods are:

- relevancy feedback

- pseudo relevance feedback (blind feedback)
- indirect relevancy feedback.

Relevancy feedback is most commonly used method. [7, p. 177] It is used after the user has given search query and receives initial results. From the results the user marks relevant and non-relevant matches which gives system feedback it can use to find similar documents and repeat the feedback gathering. This process can take multiple iterations. The method is most suitable in cases where the user does not know the system documents well. Relevancy feedback allows user to slowly work towards the goal and gain better understanding of the data. The pseudo relevance feedback is automatic process which drops the user out of the feedback loop, the system gets the initial result set but instead of asking user feedback the system blindly assumes that some count of top-most documents are relevant and uses these to refine the search automatically. This works most of the time but might skew the results in wrong direction if the initial query was not very accurate. Indirect relevancy feedback is another automatic process which uses some indirect method of gathering the feedback. For example in a web search engine the engine might use how many times user has clicked the document or web page to rank those higher in results.

3.4.4 Evaluation

To retrieve information the user of the system needs to provide a query on what information to get. After the information is retrieved we can measure quality and quantity of the process. The documents in a result set returned the search query may be classified into four groups as follows: [12, p. 114]

- retrieved and relevant (true positive)
- not retrieved but relevant (false negative)
- retrieved but irrelevant (false positive)
- not retrieved and irrelevant (true negative).

The IR process uses binary classification to determine if the document it is processing is relevant or not. The results of the classification can then be collected into an 2×2 confusion matrix as shown in Table 2, on one axis list the expected results and other lists actual results.

Table 2: 2×2 Confusion Matrix.

	Expected positive	Expected negative
Actual positive	True positive	False negative
Actual negative	False negative	True negative

The main performance measurements in information retrieval are *precision* and *recall* of the search query results. The precision represents the quality of the information retrieval, e.g. what fraction of the results were relevant for the query. It is calculated as a percentage comparing how many true positive tp matches we got out of all positive, either true positive tp or false positive fp matches. [7, p. 5]

$$precision = \frac{tp}{tp + fp} \tag{7}$$

The another main measurement is quantity of the search results or recall which is the fraction of relevant documents from all documents were returned by query. [7, p. 5]. It is calculated as a percentage of correctly identified documents tp out of all true positives and false negatives $tp + fn$ in the returned documents.

$$recall = \frac{tp}{tp + fn} \tag{8}$$

Sometimes it is easier to use single measurement for the information retrieval which combines both precision P and recall R. One option is to define *F-measure* for it:

$$F_\beta = \frac{(\beta^2 + 1)\, PR}{\beta^2 P + R} \tag{9}$$

The β parameter allows adjusting which of the recall or precision is weighted more. β values larger than one weight the recall more, while β values less than one weight precision more. Having the β as 1 weights the precision and recall equally and its most used metric and is called F_1 *score*:

$$F_1 = \frac{2PR}{P + R} \tag{10}$$

While these equations allow to measure the effectiveness of an information retrieval system they are hard to use in practice as their use require the knowledge of total amount of relevant items in the system. [18, p. 103]

3.5 Data and Information Retrieval Tools

There are many tools which implement data and information retrieval features. For this thesis the two most significant tools are relational databases and search engines to gain understanding of their basic principles and how they may be used for implementing searches.

3.5.1 Relational Databases

Relational databases or Relational Database Management System (RDBMS) are systems which store and index data and allow users to query it using relational operations. [21] The default query method in relational databases is using declarative language called Structured Query Language (SQL) [21, p. 111] The use of declarative, artificial query language and use of exact matching makes databases primarily use data retrieval methods for accessing data. The databases are commonly used as primary data stores of DAM systems because of their Atomicity, Consistency, Integrity, Durability (ACID) transaction properties offer good guarantees that data stored in them is kept safe: [22, Ch. 13]

Atomicity

All modifications in a database transaction must either happen or be reverted as a whole. It is critical that there will not be any half-applied transactions on any error situation.

Consistency

Actions in a database transaction which violate the database consistency rules must be prevented from executing and whole transaction to rolled back to previous state. This ensures that each database state follows consistency rules set in the database.

Isolation

When multiple transactions are executing they must not prevent others from executing. Also, the transactions should not see the intermediate changes of other transactions. This does not specify in which order the transactions should be applied to the database, only that they do not interfere with one another.

Durability

Database must ensure that transactions committed to database are not lost. This is usually achieved by using transaction logs. The transactions are written into the log before actually storing the transaction to the database. The database writing might be slow process so if some error happens during the database writing, the database can be rolled back to previous stable state and the missed transactions replayed from the transaction log in to the database.

The RDBMS represent data using the relational data model. In it a collection of tables is used to represent data and relationships among the data. The relational model is a combination of three components: [21, p. 67]

Structural Part The database is defined as a collection of relations.

Integrity Part The integrity is maintained by using primary and foreign keys.

Manipulative Part The database is manipulated by relational algebra and calculus.

Relational database systems use various indexing methods to improve the query execution time. Indexes are used similar to as table of contents in a book. Instead of going through the whole book to find out section about a specific topic, the reader can look at the table of contents where the section is and skip to the right place. These kinds of indexes are called *forward indexes*. [11, p. 47]

Although relational databases use data retrieval methods most modern database systems allow the use of Full Text Search (FTS) bringing some IR functionality in them. [7, p. 195]

3.5.2 Search Engines

“Search engines are one of the most widely used implementations of IR systems.” [11, p. 40] They aim to satisfy user information needs by retrieving relevant documents from their index to satisfy user search queries. The search engines may be broadly categorized into following groups: [11, p. 40]

Web search Search engine focused on indexing web content and content therein.

Vertical search Search engine dedicated on searching specific domain, such as finance or healthcare. Very specialized to their own domain.

Desktop search Search engine to indexing contents of files in users computer.

Others Search engines focusing of search beyond texts, like image, audio fingerprints or speech recognition.

The exact features a search engine provides vary but common features in them may contain following features: [23]

Indexing
: They offer many types of indices, but the most common one is the inverted index. Indices are used to improve the response times of the system.

Search-As-You-Type
: Search-as-you-type or “instant search” automatically run search while the user is typing in the search query, thus interactively filtering results.

Fuzzy search
: They provide various methods for approximate matching of search terms. These are done to cope with user making typing mistakes or not knowing the exact written form of the result.

Truncation
: Truncation allows the matching to skip parts of the text, for example so that the end of the search term is ignored when searching, effectively doing

prefix-only matching. This helps searches were user knows some part of wanted result but not the whole entry.

Normalization

They normalize the search documents so that the key terms are easier to search offering many alternatives such as down casing, Unicode normalization etc.

Match highlighting

Search engines might offer methods to see where in the document a match for user query was found. This is for providing good user experience, so the user of the system can visually see where the match occurred.

The above lists quite basic features in search engines, but they commonly have many features and options to fine-tune them for each search case. Some of more advanced search features supported might contain following: [11, p. 209]

Sponsored search

Aim is to boost given documents so they appear on the top of search results or alternatively exclude some matches from results. This can be used for quickly applying fixes for production issues as well by hiding the wrong results until a proper fix is implemented.

Spell-checking

After user has typed a search term, the system might automatically offer to autocorrect it. For example mistyped word is automatically replaced by a correct term if found in the indexes.

Autocomplete

also known as type-ahead search where the search engine offers possible completions for the search term user is typing and the list of offers is refined after each new character. Benefit of using autocomplete is that it avoids spelling mistakes and reduce the effort to use the system as users will not need to type full search terms themselves. This also allows implementing system provided recommendations by offering better search terms.

Document similarity

They can offer “More-like this” documents in addition to regular search results.

3.6 Metadata

A document may contain additional information besides its content. This is called *metadata* which may be defined as “data about data”. [18, p. 9] Metadata is commonly used to give document creation times, author, perhaps the program name used in creation of the document. This data may be embedded in the document itself or stored separately from it. [18, p. 7] Working with metadata is made difficult by the lack of unified standard which makes interoperability between programs working with document harder. The rise of Internet in mid-1990s gave a need to have language- and discipline agnostic metadata standard to categorize and enhance an information retrieval for web content [18, p. 7]. Metadata standard called Dublin Core Metadata Initiative (DCMI) was created to improve search and discoverability of digital information resources. [18, p. 95] The DCMI may be extended to cover the needs of specific domain by creating custom profiles. The use of well-established standard allows easier data exchange of data and aids making the conventions and intentions used with metadata clearer [18, p. 49].

There are many purposes for the need of metadata and the Haynes used following six key points in his book Metadata for Information Management and Retrieval: [18, p. 15]

1. Resource identification and description
2. Retrieving information
3. Managing information resources
4. Managing intellectual property rights
5. Supporting e-commerce and e-goverment
6. Information governance

In context of DAM the first four points are most relevant so the last two are omitted. Any information management system requires a method to accurately identify things. [18, p. 78] This usually means adding some uniquely identifying metadata field for the resource such as Universally Unique Identifier (UUID) or similar entry. Another important role with metadata is to provide accurate description for the information resource, who made it (author/creator), when it was made (date of creation), what it's called (title) and any additional information (description). [18, p. 78]

Accurate metadata is also essential in providing adequate search for resources. [18, p. 96] Metadata may be used for adding semantic context for resource which can be used by search engine and application and use of metadata standards such as DCMI aids in this process. [18, p. 104]

One problem area for search is multimedia records, such as images and videos as they are not composed of text. [18, p. 108] One method for having information retrieval on multimedia would be adding face recognition, speech recognition and similar processes to enrich the metadata.[18, p. 109] This would allow use of IR methods described further in Section 3.4 to retrieve records based on the keywords. Alternative would be to use methods such as Content-Based Image Retrieval (CBIR) [24] which analyzes query image to find visually similar content. [24, p.2]. Both of these require ML which was described in Section 3.1.

Metadata should be utilized at each step of the record lifecycle. Haynes presents simplified information record lifecycle consisting of following steps: [18, p. 115]

Creation of information The step when record is created, essentially capturing the content and attached metadata.

Distribution and use The created records are used for their purpose, either internally or by distributing them publicly to users.

Review Information contained in a record may become stale, so it needs to be reviewed to keep it accurate.

Preserve and store The record needs to be stored securely even as the technology changes.

Dispose Once record becomes obsolete it should be disposed from active use, either by deleting or archiving it to long-term storage.

Transform Old records may be used as base on creating a new record, starting the lifecycle from the beginning.

To manage intellectual property rights the record metadata should contain following fields at minimum: [18, p. 128]

- Name of the author
- Time of creation
- Copyright associated with the work
- publication status
- Date that rights research was done

As demonstrated above, metadata has huge impact on providing accurate information retrieval methods in a system.

3.7 Digital Asset Management

DAM is a system that stores, manages and distributes digital assets in a controlled way. They may do this as single system or in combination with other systems. [25, Ch. 1] DAM aid in reducing search cost and digital preservation of assets [25, Ch. 2]. A digital asset is any digital data which has rights to use attached to it [22, Intro]. Public data without any usage restrictions is not considered an asset. Different organization teams might each have their own system to manage their assets creating *information silos*[25, Ch. 2]. These silos make it hard to figure out which version of assets to use and causes organization to use many redundant systems for asset management. [25, Ch. 2] They also increase effort for locating wanted assets, each system needs to be known and searched separately. DAM aid in this by providing centralized system to ease finding the relevant information. [22, Ch. 9]

The DAM systems do not seek to replace other tools used in enterprises, rather the aim is to have the DAM system to glue different applications together. [22, Ch. 6] The typical components included in a DAM system include following:

- Content repository

- Digital asset management application
- Databases
- Search engine
- Indexing workstations
- Rights management application
- Web portal

The content repository is place to store the asset files, be it local file system, network share, or cloud storage. One commonly used option for storing the asset metadata is to keep them in relational databases. [22, Ch. 13] They are mature and stable systems so provide operational safety. The original relational model was extended with object-oriented approach creating *object-relational databases* allowing them to store object which allow storing mark-up files such as XML data directly to database [22, Ch. 13]. The XML is commonly used when working with metadata so to be able to directly store it into database works well for DAM use case. [18, p. 19] Search engine is used to index the asset metadata and contents for providing accurate search. Indexing workstations might be required if the storing of asset requires considerable processing, in cases like transferring analog film content into digital form. Rights management would be used to have Digital Rights Management (DRM) features for the assets. These could be utilized to while distributing assets so that once they expire they would no longer be available. The web portal is used for providing access to system functionalities and finally the DAM application itself would be in the background managing all these components.

The DAM systems can be categorized based on what kind of assets they are managing. Common subsets of DAM systems may include following: [25, Ch. 1]

Brand Asset Management (BAM) tailored help company branding, what are the latest brand images to use etc.

Production Asset Management (PAM) is tailored to tracking frequently changing assets.

Library Asset Management (LAM) tailored to cataloging fairly static set of assets such as videos or documents.

Media Asset Management (MAM) tailored to handling media files; images and videos.

Document Management (DM) System optimized for handling of documents and to some extent media files, for example the needs of a legal and human resources.

Enterprise Content Management (ECM) DAM system containing links to many other systems to form larger system.

To make matters more complicated there exists systems that offer similar features as DAM but not being part of it [25, Intro]. These are Content Management System (CMS) and Web Content Management (WCM). These tools are meant intended for web page creation and management, and they do provide long-term asset storage and sophisticated search features even though they otherwise are similar to DAM applications. [25, Intro]

The DAM Foundation as cited in [26, p.15] define following features which are exhibited by DAM systems:

Asset Ingestion Assets may be added into system individually or *en masse*. The system handles identification of an asset by use of unique identifiers.

Asset Security DAM system employ Access Control List (ACL) to control which users may access assets within the system.

Metadata Assets are stored as data files containing the asset contents along with the metadata defining additional information. The system helps to manage the metadata.

Transformation The system allows transforming assets to various formats in automatic methods, such as creating thumbnails or converting images to different common formats.

Enrichment The system help enrich the asset metadata by collecting statistics on their use throughout asset lifecycle.

Versioning The DAM systems commonly include versioning support to track how the original asset has evolved in the system.

Workflows The systems may define workflows to aid users in creating, managing and process assets in the system.

Search DAM systems employ powerful search methods to increase discoverability of assets.

Previews Previews are used to increase productivity, users may quickly scan search results for the correct version instead of downloading full version of each compared asset.

Publishing The assets stored within a DAM may be shared to users outside the DAM in controlled manner.

The users of a DAM are commonly using one of three kinds of search methods, navigational, direct or faceted search. The *navigational search* is the most basic one where user browse categories and click through assets to see what they might contain. In *direct search* the user is executing a search query and examining the query results. The *faceted search* extends the direct search by having the user using facets or groups to further limit the results by those by facets such as color or file type. [25, Ch. 8]

4 Project Specification

This chapter describes the details of the new DAM application. From technical perspective it presents the system's current architecture, how it is built and what kind of features it has. After examining the new system the chapter moves on to describing view points which are to be used to analyze the system starting with the results of project management interviews. These define project management's views on how the search features of the system should be improved and what would be the initial scope for them. The next view point is from the users of the system. A customer survey was conducted to gather data on what search features would be most needed in their work. Next the view point is from past user behavior by analyzing server logs from old system to see what kind of searches are used in it. As the design goal of the search feature was to be "simple as Google" this research includes a brief analysis on what this could mean by looking at the features in Google search. Finally, the chapter includes a review of the search related features present in the major DAM providers, what others have found to be beneficial features on asset searching.

4.1 System Overview

The DAM application in this case study is a new system developed to replace old, more complex system. The new application architecture follows the microservice design where the system consists of many individual components which communicate with each other through some predefined protocol, in this case, by using Application Programming Interface using Representational State Transfer (REST-API) over Hypertext Transfer Protocol (HTTP). The benefit from design over traditional monolithic design is the loose coupling which allows each part to be worked on separately, possibly by separate teams. Communication between instances is done through the Application Programming Interface (API), so any internal changes do not need to synced between the teams. This separation also allows redesigning part of the application without huge refactoring of the whole system. The DAM service is self-hosted using Kubernetes, an open-source tool for automating the management of containers originally developed at Google[2]. The system was designed using self-hosted solutions as various customer agreements at the design time prevented from hosting the system on an external cloud providers.

The system is developed using TypeScript[3] programming language which is a typed superset of JavaScript. The language adds strong typing support with aim to help developers avoid many runtime issues with the applications as these are caught during the build process by the type system.

The overall architecture is presented in Figure 7. It follows that of the multitier design where the system has a presentation tier to offering User Interface (UI) for user interaction, application tier handling the business logic and controlling accesses and finally the data tier where the actual system data is stored. [22, Ch. 6]

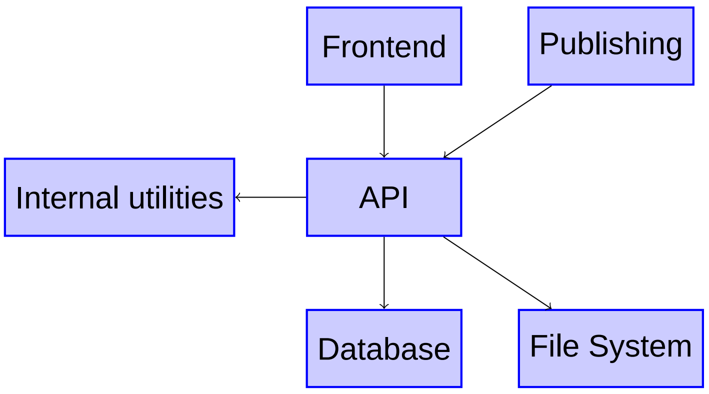
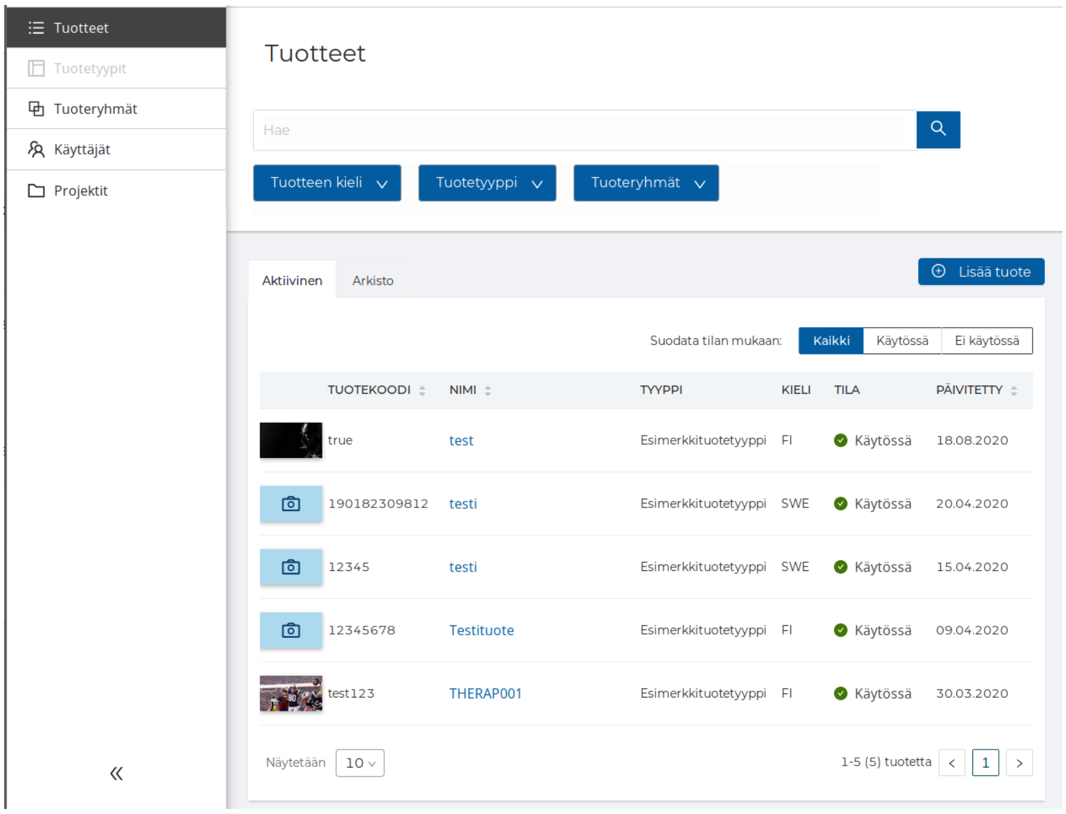


Figure 7: Overview of New DAM System Application Architecture.

The presentation tier consists of a frontend and publishing applications. The frontend is a React-based application which runs on the client browser and provides users with easy-to-use interface to work with the DAM presented in the Figure 8. A sample of The frontend passes user actions to the backend API to make changes in the system.

Other component on the presentation tier component is the publishing application. The application is used by the DAM to publish or share access to any give asset.

[2] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[3] https://www.typescriptlang.org/



	TUOTEKOODI	NIMI	TYYPPI	KIELI	TILA	PÄIVITETTY
	true	test	Esimerkkituotetyyppi	FI	✓ Käytössä	18.08.2020
	190182309812	testi	Esimerkkituotetyyppi	SWE	✓ Käytössä	20.04.2020
	12345	testi	Esimerkkituotetyyppi	SWE	✓ Käytössä	15.04.2020
	12345678	Testituote	Esimerkkituotetyyppi	FI	✓ Käytössä	09.04.2020
	test123	THERAP001	Esimerkkituotetyyppi	FI	✓ Käytössä	30.03.2020

Figure 8: Product Search on the New DAM System Frontend.

This is done to give unauthenticated users limited access to some specific version of an asset. Each publishing application is tailored individually to each customer, but they share same general principle.

The main functionality and business logic of the system is done on the application tier. The tier is centered on the backend application which provides the system REST-API for the frontend and any other connecting services. The application tier also includes many internal components which are only accessible by the API. The application tier is the only one which has access to the system data. The data tier is split between shared network drive storing the asset files and a RDBMS storing the asset metadata. The RDBMS used by the new system is PostgreSQL[4] which offers several extensions to the standard SQL which are leveraged by the new system, particularly the FTS [27, p. 441].

The DAM system in under study can be classified as a mix of a Production Asset Management (PAM) and Document Management System (DMS) as it offers

[4]https://www.postgresql.org/

following features:

Role-Base Access Control Access to assets and system functionality is controlled by Role-Based Access Control (RBAC).

Workflow Users of the system may collaborate in drafting a new asset.

Versioning Assets stored in the system are automatically versioned.

Change history The system tracks change history for each asset.

Grouping Assets can be manually placed into groups.

Searching Assets can be searched using free-text search and/or filtered by group, type or language.

Distribution Assets can be published immediately or scheduled for later publishing.

Metadata management system allows users define templates for metadata for each asset type.

The system is targeted to domestic B2B market, which means it needs to support three languages: Finnish, Swedish and English. The system is offered as a SaaS application for the customers. The system multi-tenant meaning a single instance is providing service for multiple customers.

The data amounts handled by the system have been quite small, staying in the gigabyte range, but this is expected to grow rapidly into hundreds of gigabytes in the near future. From production asset management side the application offers workflows. The workflows allow users to collaborate in drafting a new asset. These drafts can be started from scratch or by using existing asset as a template.

Currently, the assets managed by the system include mostly text documents and some image files. The assets in new system are represented by products and renditions. A product represents a single top-level asset information and each product has the following information:

name the name of the product.

type the type of the product, which is used to set the metadata fields of the product.

description free-text description of the product.

product code user specified product code identifying the product, may be auto-generated by the system.

content language the product content language.

metadata Data giving out product type specific metadata, publishing unit, these use customer given rules.

created by who created the product in the system.

The above lists the shared properties present in each product. Each product also has type which defines template for what kind of metadata may be set for given product. The template defines which additional fields of metadata are mandatory and optional on each product using this type. In addition, each product may have renditions. A rendition or version represent the file asset managed by the system. Renditions store similar kind of metadata as products. The renditions use rendition type which specifies what metadata is available for given rendition.

rendition type type of rendition, sets the allowed metadata.

product id which product this rendition belongs to.

created by who created the rendition in the system.

version id user given version identifier.

published timestamp when the rendition was published.

expired timestamp when the rendition was expired.

version num internal version number to track the latest renditions.

file path internal file path where rendition data is found.

file size size of the rendition file.

external filename the external name for the rendition file.

metadata the rendition metadata, determined by the rendition type.

Certain rendition types allow distributing the rendition asset in process called publishing which allows giving access to the given rendition within some publishing channel. These are tailored for each customer as each of them have different publishing needs, although the general process is similar in each case. The user is prompted to give time range for when the rendition status should be public when saving a rendition into the system. These are controlled by setting published and expire date fields when saving a new rendition. The system will use these dates to handle the technical side of publishing when the time is due. The default method for the publishing is to mark the rendition public and have the systems publishing component to access the rendition.

There exists two methods on making assets public in the system. The first one is simple link-based publishing where the backend API component generates a link leading to the publishing component. Accessing the link allows anonymous user to download the asset file while the link is marked public by the system. The system may generate two kinds of these links, either direct links to specific rendition or *a floating link* which always returns the latest public version of rendition.

Some customers use more customized publishing methods where public assets are instead transferred to customer intranet application for their own internal processing pipelines. For others the publishing component offers simple web application providing form-based UI where external users may access, browse and search through public renditions.

4.1.1 System Search Features

The main search methods of the system are provided by two API endpoints:

- /search/products
- /search/productsWithRenditions

Difference between the endpoints is that first one searches for matches from all products in the system. The latter limits the search to only those products which have at least a single rendition. It also extends the search options by allowing to limit the results by rendition status which allows free text search for products which have published renditions. Both of the endpoints offer following search options to limit the results:

keywords Filter search by given free text query.

product type id Filter search to specific product types.

language Filter search to specific languages.

product groups Filter search to given product groups.

parent group Filter search to those products belonging to given group or any of its subgroups.

owner Filter search to products owned by given user.

The user of the DAM system may trigger the asset search process from three places: the search bar in frontend application, same in publishing application and accessing the system REST-API directly.

The search query transformation is happening on two parts of the system. The first one is done by the frontend which transforms the user's frontend component manipulations into a REST-API call on the backend component. The second query transformation is done when the backend transforms the REST-API call parameters into a SQL query to get the information the user has requested from the database.

The current search uses mix of information and data retrieval methods. The free-text query entered in search bar triggers a full text search in the database and various filters use data retrieval options to restrict the full text search to a known subset of system data.

One of the problems in the system search is that there is single search interface but customer use-cases for it are different. The systems history is with document management where customers have used the system to version and design their internally and externally shared documents. The workflows in this have relied heavily on each asset to having their own unique product code which is used to sync assets with other systems. Newer customers have more media-oriented needs, they use the DAM to store images and video. Their search requirements focus more on the content of the assets themselves, for example searching assets containing happy people with laptops in them. Difficulty is how to keep the search general enough but still able to work with each customer workflows.

4.1.2 Issues in Current Search

The new system exhibited some issues in the implemented search feature at the beginning of this study. These were uncovered by actual system users in their daily work and were reported to the project management. Most of these were inquiries on why the system did not return the expected result for given search query. The user reports were analyzed and they could be grouped into five different issues in the system.

The first issue was that the database trigger function indexed wrong terms. The trigger handled the username field so that '@.+' characters in it are replaced by space character. The system was designed so that users email address is used as usernames, so they are usually given in format *first_name.last_name@company-domain.fi*. The trigger attempted split this into separate fields, so search can be done by first or last name. This had the side effect that the search using full email address would not return matches anymore as the full email address was no longer indexed and the same modification was not done on query time.

The trigger also used "simple" dictionary to parse all fields. Dictionary is a set of configuration options which specify what operations are used during parsing the field into a tsvector. Using "simple" meant that the system was only doing minimal normalization to the input, effectively just lower casing it before storing it to index.

The search parser was noticed to have few problems. First one was that when users gave multiple search terms in the search field the system did not return all the expected results. Examining this more closely this was caused by flawed implementation shown in Listing 13 which effectively makes the application search to do phrase searches by default and use prefix matching only for the last term which does not seem what the author has intended.

Other related issue was identified when searching only product codes where giving only part of the code did not return match. By looking at the executed search queries and their parsing more closely it was related on the database parsing error causing hyphens leading a number to be parsed as a negative numbers leading to incorrect results in some cases.

Final issue in the search is also related to the product code searching which is commonly used search term by users. Users need to query product codes such as “ABC 2/2.10”, but these are currently not found by the search implementation.

Table 3: Summary of Known Issues with Current Search.

Issues
email searching is not working as intended
non-optimal indexing using “simple” dictionary
phrase searches done incorrectly by default
product code parsing is not consistent
product code matching with “special characters”

The Table 3 presents identified issues in the new DAM search at the time of the thesis. The search improvements should focus on fixing the above while providing the wanted features for improving the user experience for the system users.

4.2 Views on the System Search

Besides just fixing existing issues the primary aim of the study was to identify what features would be required to be in it. Research was done to gain better understanding what features they would be, gathering information from various view points. The first view point was from the current project management by doing an in-depth interview with them.

The interview was done using qualitative approach to get wide overview of whole system state. The questions covered in the interview are listed in Appendix 1, but the discussions spread to other areas besides the listed questions during the interview. The aim of the interview was to gain insight on what were the current issues and how the management saw that the application would be used in the future now that they and customers had some experience on using the new system.

The search features of the new system were not specified accurately during the initial system design. The project management gave rough overview “the search must be fast, simple to use and should cover all data” and left the practical technical implementation to the subcontractor. The subcontractor did not question the description and project management did not think they would have needed to specify the minute details of the search. The current system search is based on the vague original requirements which has seen some small improvements since.

In addition to this in-depth interview a user survey was conducted. The survey was done using quantitative approach and the survey questions can be found in Appendix 3. The customer key users were asked for consent for the survey to be done and to get up-to-date user lists who to invite partaking in the survey. The invited users were given approximately three weeks to answer the survey. The survey was done anonymously unless the user agreed to be interviewed by giving an email at the end of the survey. The aim of the survey was to gather users views on how well the new systems features worked for them and what kind of search features they felt were useful in their work.

One additional view point was to look up past user search behavior by analyzing old DAM system server logs to see what kind of searches have been done in the past. Unfortunately the server log rotation had already deleted old logs which would have shown past behavior of the customers which were already migrated into the new system. But there are few larger customers still using the old system waiting for migration. By looking the searches of all remaining customers the free-text search query commonly contained only a single term. This was either a product code or a single word. These two options seem to cover roughly 90 percent of the queries seen in the logs. Rest of the queries were divided between phrase searches like user has copied and pasted the product name if it is composed of multiple words or by using multiple search terms. These search terms seemed to be simple words mentioned in the document, for example “form”, “cat” etc. There have been studies on user search behavior [28, p.180] on search engines showing that most of the time users search queries contain only two terms on average. The findings from old system appear to validate the analysis results of the previous studies.

The next view point was to analyze Google web search as it was one of the original design goals of the search. The aim is to define what “Google-like” search would mean. The Google web search is minimalist and simple in its design as shown in Figure 9. It presents user with “a large search box and the simple, uncluttered user interface” [29, p. 62]. Once Google has processed search query given by user, the results are listed below the bar. Second distinguished feature is the search speed as results for search query are determined in well under one second. [29, p. 63] The results themselves are ranked by relevance. The results also include small amount of context where the match was found along the returned Universal Resource Locator (URL). This simple UI hides the complexities of the search mechanics running in the background which today include sophisticated NLP and ML technologies. [30] The search engine has the largest

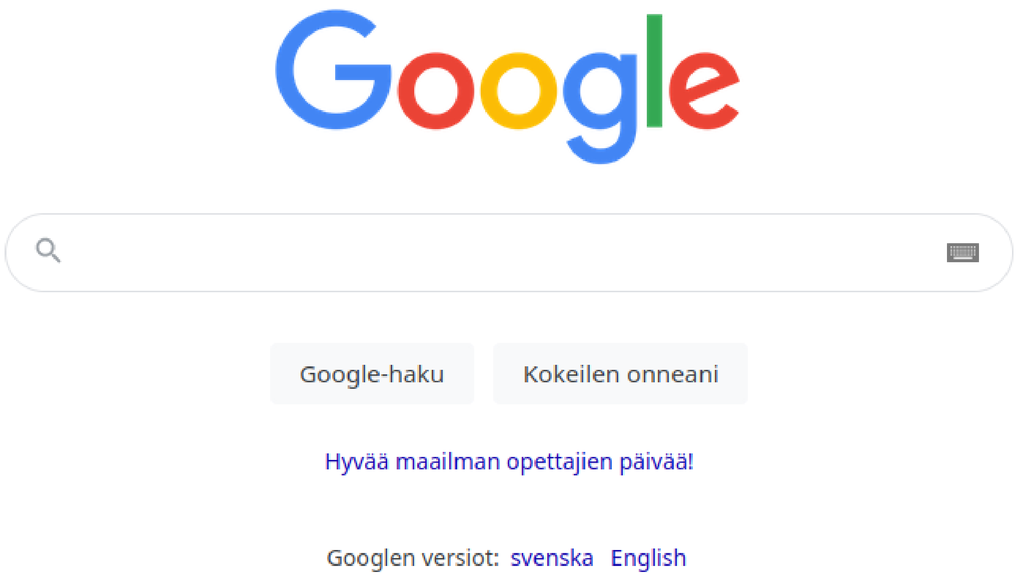

Google
Google-haku
Kokeilen onneani
Hyvää maailman opettajien päivää!
Googlen versiot: svenska English

Figure 9: Example of Google Web Search Page.

market share[5], so making the DAM search work in similar manner would make its use familiar to a wide audience. Besides the search bar, the Google’s search may also be used with more fine-grained filters when using the advanced search[6]. This simple search interface, keyword search, and separation of quick and advanced

[5] https://www.netmarketshare.com/search-engine-market-share
[6] https://www.google.com/advanced_search

search are directly transferable to DAM system even though they operate in different domains.

Last view point in this study is to look at the current market leaders in the DAM market. This was done to help see where market seems to be heading instead of just identifying and fixing issues in current search implementation. These larger systems show what features have been seen as beneficial in DAM systems, so it helps to build a road map for future improvements.

Table 4: Summary of View Points to System Analysis.

View Point
Company requirements in form of project management interview
Feedback from current users by use of survey
Analysis of past user behaviour from old system logs
Defining more accurate definition for original “Google-like” search feature
Market analysis by examining DAM market leaders products

5 The Requirement Specification

The aim of this chapter is to present the findings of view points in Table 4 and use them to generate requirements on what kind of search features the DAM system should provide to serve current and future needs.

The first and most important views for system requirement came from project management. They were interviewed using open questions to cover wide range of ideas for the DAM search. The results of the interview are presented in the Appendix 2. Comparing each interviewee's views showed that the management did not have unified goals for the search improvements. This study focuses on features all interviewees agreed were essential to have in the system. The interviews covered many topics but the Table 5 collects priorities of the uncovered main points. The table omits the column for "wish it had" features as it did not have any values.

Table 5: Summary of Project Management's Wanted Features.

Feature	Must have	Should have	Could have
Sub-second search times	x		
File metadata indexing	x		
File content indexing	x		
Image recognition		x	
Speech-to-text		x	
Machine translation		x	
Extended search scope	x		
Free text search	x		
Truncate search	x		
Multiple search terms	x		
Phrase search		x	
Relevance ranking	x		
Autocomplete			x
Autocorrect			x
Search-as-you-type			x
User search history		x	

The Table 5 demonstrates that management roughly want the system to include keyword-based FTS and extend the material that should be searchable. Next steps would be to extend the search to multimedia files such as images and videos. Once these would have been implemented the search could be extended to cover more advanced searches such as autocorrect.

Based on the interview findings, a customer survey was created and can be found as Appendix 3. It was sent to those system users which had been using the new system for some time to gather their views on how the search feature should function. Based on the prioritization definitions given in Table 17 the required features based on the customer survey results are collected in Table 6.

Table 6: Summary of the Customer Survey Results.

Feature	Priority (total)	Priority (positive)
Wildcard or truncate search	Must-have (14)	Must-have (14)
Free text search	Must-have (14)	Must-have (15)
Ranking of search results	Must-have (10)	Must-have (11)
Inflection support	Should-have (7)	Should-have (8)
Search filters	Should-have (6)	Should-have (10)
Highlighting	Could-have (5)	Could-have (8)
Smart tags	Could-have (3)	Could-have (7)
Search-as-you-type	-	Could-have (6)

When looking only the positive scores it does not make sense to use same prioritization as there are no negative scores, instead the Table 6 positive column lists only top features with scores of 5 or higher. As the table shows the scoring method did not impact the end result in a meaningful way. As the results in Table 6 show, customer had quite modest requirements regarding the search. Highest priority was to have keyword-based search with some users wanting to have smart tagging with AI included.

Another view point on how to improve the search feature is to look what other vendors are doing. This gives information on what is essential to implement to stay competitive in the market. For choosing which vendors to focus on an intersection was taken from vendors analyzed in market research done by

Gartner [31] and Forrester [32]. The reports categorized different DAM providers into groups based on their market share which will also be used in this thesis giving the vendor list in Table 7.

Table 7: Classification of Existing DAM Vendor Products.

Product	Classification
Northplains Telescope	Challengers
Digizuite DAM	Contender
Canto	Contender
Widen Collective	Strong Performer
Nuxeo Platform	Strong performer
CELUM Content Collaboration Cloud	Strong performer
Bynder Flagship	Strong performer
OpenText Media Management	Leader
Aprimo DAM	Leader
Adobe Enterprise Manager Assets	Leader

The details presented here for each DAM product are based their vendors own marketing materials and documentation which were available from their website. The reviewed systems are proprietary, and they did not offer a live demo environment for hands-on testing. The review was done at the 17.10.2021, so the exact details might have changed since but the overall themes should be the same. The available documentation varied by each vendor, some vendors provided very accurate and detailed documentation, others had only marketing information available and reserved the documentation for paid customers only. This is not a problem for this study, this section is here to list general search themes in the largest DAM vendors making exact details not that important.

The Forrester report [32] lists three main trends which are present in successful DAM provider:

Simple and intuitive user interfaces The systems should be usable by users of various technical backgrounds.

Integrations Instead of keeping the DAM a “data silo”, allow users access the assets within from various tools by integrating them with it.

Enriching content Instead of keeping static assets, enrich the assets with embedded or external AI solutions, adding smart tags or allowing AI guided image editions directly creating new assets in the process.

All the above give good guidelines on where the DAM solutions should be heading. Details of the vendor analysis are provided in the Appendix 5.

The feature list in Table 18 is the same which was used in the customer survey. To narrow the features to more manageable list a summary presented in Table 8 containing only features which were present at least on half of vendor products.

Table 8: Summary of Most Common DAM Features in the Top 10 Vendor Products.

Feature
Truncate search
Search filters
Free text search
Support multiple terms
Phrase search
Word inflection support
Result ranking
Dynamic search facets
Smart tags
Search should include file contents
Quick and advanced search

The summary supports the views on the project management and customers that keyword-based FTS appears to be the most used feature. In addition to keyword search the vendor products provided advanced search features such as dynamic faceting, smart tagging. Search was able to look at file contents as well as system metadata supporting the wishes of project management in Table 5.

5.1 Defining the System Requirements

To specify the requirements all view points should be considered. The user survey should indicate that the users are interested in keyword searching with pattern matching and ranking of results. The next wanted feature was to have the system accept close enough search terms to give matches. This could be interpreted as fuzzy matching or stemming. Simple search filters are already provided by the system which seems to be enough. Some users expressed their need to have highlighting where the match was found, as the indexed data increases this might be harder to implement and would require more detailed specification. Last feature for which users expressed interest on having was AI-powered asset tagging. This would be used to reduce manual work done in the system.

Table 9: Requirement Specification for the New DAM System.

Feature	Priority
Sub-second search times	Must-have
File metadata indexing	Must-have
File content indexing	Must-have
Extended search scope	Must-have
Free text search	Must-have
Truncate search	Must-have
Multiple search terms	Must-have
Relevance ranking	Must-have
Search filters	Must-have
Phrase search	Must-have
Machine translation	Should-have
User search history	Should-have
Image recognition	Should-have
Speech-to-text	Should-have
Inflection support	Should-have
Smart tags	Should-have
Dynamic search facets	Should-have
Autocomplete	Could-have
Autocorrect	Could-have
Search-as-you-type	Could-have
Highlighting	Could-have
Quick and advanced search	Could-have

These seem to align with the project management wishes and customer survey results. The most common features are FTS which can cover file contents, sometimes split into separate quick and advanced searches to skip file content searches by default and using advanced search to cover the contents. Many provided automatic asset tagging and dynamic search facets on the results. Collecting all these findings together we get the required features listed in Table 9.

To make it easier to work with the features from Table 9 this study categorizes them into few groups presented in the Table 10.

Table 10: Classification of DAM Requirements.

Grouping	Search Features
Constraints	fast search times, metadata and content indexing, extended scope
Keyword Search	free-text search, multiple terms, truncation, ranking, phrase search, inflection, highlighting
AI Enhancements	Machine translation, image recognition, speech-to-text, smart tags
Advanced Searches	Dynamic facets, autocomplete, autocorrect, search-as-you-type
Miscellaneous Features	Search filters and history, quick/advanced search

The first group of are not features themselves but set constraints mostly coming from project management to keep in mind while implementing new functionalities. The other groups specify more concrete feature sets, keyword search is the most important group of features to implement in the DAM so most of the analysis in Chapter 6 is focusing on this. The AI features group has the features which build upon AI and ML to enhance the user experience of the system. Advanced search group lists common functionality found in search engines which might be tricky to implement with just a relational database such as PostgreSQL. These were not strictly needed, but each improve the user experience of the system. The last group is the miscellaneous features of which search filters are already provided by the system.

6 Search Implementation Analysis

The goal of this chapter is to analyze if and how the current architecture using the PostgreSQL database can fulfill the requirements listed in Table 9. There are few points in favor to keep using the existing database for implementing the application search. First is avoiding the need for installing and maintaining extra software unless there is compelling reasons to use alternatives. [22, Ch. 13] The second point is that the application can keep using the same interface for search as before, which is currently SQL. Last point in favor is that by using single data store gives the data a single source of truth. Using external tools would require syncing the indexable data between the database and search tool which leaves opportunity to them to go out of sync. The databases ACID properties help keep unified view on data but this does not extend to data stored outside of it. The use of external system along the database would require to keep user permissions in sync between them, so that users with limited data access cannot access data their permissions normally would not access to, through the use of external search engine.

6.1 Inspecting Constraints

The project management defined few high-level constraints which greatly affect what kind of search solutions may be used in the new system. One of the features the project management wished the system to handle was to store and index the file metadata and content in addition to system provided metadata. As noted in earlier Section 3.6 metadata plays huge part of search implementation making this high priority task. One way to achieve this is to use Apache Tika[7] to parse files when adding them to the system. Tika is a Java-based tool which can parse several file types and extract metadata and file content from them.

The available metadata in files varies by file type. For example a plain Javascript Object Notation (JSON) file does not provide anything particularly interesting

[7] https://tika.apache.org/

metadata shown in Listing 3, just a content encoding and content length. More complex file types such as Portable Document Format (PDF) provide a great deal of metadata which contain many fields relevant for searching as detailed in Listing 4. Problem with this sample is that it contains numerous fields which are not relevant for searching, for example fields detailing how many characters are in the page 30 of given PDF document. Other commonly used file types in the DAM are documents used with Office family of tools. These are the most common file types in the new system after PDF, so it needs to have good support. These files appear to provide less metadata than PDF files as shown in Listing 5. The metadata fields of office document seem to be more relevant for a text search. Current customers are using PDF files by large margin in the new system as shown in Table 11.

Table 11: File Counts for Different File Types.

File type	Count
pdf	17074
docx	550
doc	395
zip	75

The Table 11 lists the count of different file types in the sample database. The results are filtered to show only file types of which there were more than 50. As shown the most common file by great margin is the PDF file type, rest are mostly covered by Office files. These were also commonly supported by DAM vendors as pointed out in Table 8, so system should focus on having good support for these file types.

6.2 Keyword Search

This section covers detailed analysis related to keyword search which was identified as key feature in Table 9. The PostgreSQL database includes many options for searching, from simple equality tests, to pattern matching and more advanced options such as fuzzy matching and FTS.

PostgreSQL has extensions which allow more advanced searches to be done, such as phonetic match algorithms [27, Appx. F. 15] but these are not really suited for implementing keyword search as the algorithms are intended on matching short strings such as peoples names. [7, p. 63] These might not be helpful for keyword searches but may be used to aid other searches or providing options such as autocompletion for product owners. The extensions most suited for implementing keyword search would be either the fuzzy matching provided by pg_trgm contrib [27, Appx. F. 31] or the FTS [27, Ch. 12].

The pg_trgm contrib module provides methods on using n-gram matching, more accurately trigram matching for inputs. It would fit in customer requirements of having keyword search with ranking. The added benefit of this method is that it would allow making small typing errors in the input which was seen as nice addition by project management in Appendix 2. The trigram matching provides utility methods for using it to match longer texts instead of single words making it usable in keyword search. In the previous DAM system the users were commonly searching by using single search term as presented in Section 4.2 which would make fuzzy search a viable alternative for the more complex FTS.

The PostgreSQL database extends the default relational model by providing a FTS which is useful in cases when full set of search engine features are not required. It was developed by Oleg Bartunov and Teodor Sigaev in order to handle “online updates with access to metadata from the database” [33, p. 11]. The implementation is managed with SQL and comes with extensive set of configuration options and stemmers to aid setting it up. [33, p. 12] The FTS processes the user search query, finds matching documents and returns them to the user ranked by relevance. It does this by using two data types, *tsvector* for representing stored documents and *tsquery* for search queries. [27, p.168] The tsvector type is a “sorted list of distinct lexemes, which are words that have been normalized to merge different variants of the same word.” [27, p. 168] Dictionaries are language-specific and PostgreSQL provides them for many of the most commonly used languages including Finnish, Swedish and English. Typical normalization steps include down casing words, removal of stop words and stemming. The document in this context is the fields we want to search from the database which might be from a single database table or combination of fields from several tables. The terms in tsvector may be attached with optional positional information to specify where in the document it can be found. The implementation is using ranked boolean retrieval as the terms may include different weights which affect the ranking of search results. The tsquery represents user search query and accepts set of terms to be queried joined by one of boolean AND,OR,NOT or FOLLOWED BY operator. The FOLLOWED BY operator allows tsquery to do proximity matching. Each search term may include `:*` suffix to specify prefix-matching for given term. These features implicate the FTS uses the extended Boolean model which was detailed in Section 3.4.2.

The tsvector contents may be indexed with GIN and GiST index types. This is based on the RD-trees presented in paper “The RD-tree: An index structure for sets” [34]. The indexing changes the representation of the tsvector to use *a single bit-signature*, a so-called superimposed signature is based on the index structures defined in article “‘Index structures for databases containing data items with set-valued attributes’” [35] essentially turning the set of words into single bit signature.

The database documentation [27, p. 451] lists some limitations in the implementation. The implementation is sufficient on most parts but the 1 megabyte limit on the length of tsvector may be problematic when indexing large documents with many unique terms.

One problem in fully utilizing FTS is caused by the complex morphology of the Finnish language which uses small core vocabulary which is extended by using many inflectional forms [36, p. 2]. Other problematic aspect is the languages heavy use of compound words which makes information retrieval hard as user would need to match all preceding word components when searching. The Snowball stemmer algorithm which is used in the PostgreSQL does not support word separation for compound words which would increase matching in queries. [36, p. 2] There exists a library to implement morphological analysis for

Finnish language called “libvoikko” as part of the Voikko project[8]. This was originally part of the Hunspell spellchecker, but the features needed to make it work well with Finnish language were not good match for Hunspell project so Voikko was forked as separate project to provide morphological utilities for working with Finnish language. [37] Actual extension providing Voikko lemmatization and compound word separation support for PostgreSQL is called dict_voikko[9].

The available search methods in the default installation of PostgreSQL database are summarized in the Table 12.

Table 12: Summary of the Available Search Methods in the PostgreSQL Database.

Search Method	Matching	Method	Indexable	Language dependent
= (equality comparison)	exact	-	yes	no
LIKE, ILIKE	exact	pattern	no*	no
SIMILAR TO	exact	regexp	no*	no
POSIX Regexp	exact	regexp	no*	no
Levenshtein	fuzzy	edit distance	no	no
Soundex	fuzzy	phonetic	no	yes
Metaphone	fuzzy	phonetic	no	yes
Double Metaphone	fuzzy	phonetic	no	yes
Trigrams	fuzzy	n-gram	yes	no
Full text search	fuzzy	Full-text search	yes	yes

LIKE, ILIKE and regexp operations can be indexed using pg_trgm contrib

6.2.1 Keyword Search Analysis

The database analysis was done on PostgreSQL version 13.3 with default configuration. The database was filled with database dump from new DAM systems staging environment dated 20210701. The test queries are run on single customers’ database schema. The sample database schema contained 5279 product entries and 18120 rendition entries in it. Before running analysis queries a few preliminary steps were done to check limits and prepare the tables to perform well.

For testing the limits of FTS the PostgreSQL manual [27] for version 12.4 was

[8] https://voikko.puimula.org
[9] https://github.com/Houston-Inc/dict_voikko

used. Its text content was extracted with Tika and stored into tsvector column. The manual has file size of 12.4Mb, and it was chosen as a test document as it contains 2742 pages of mostly textual content. Larger PDF documents will likely have embedded images and such raising the file size but which will not affect the indexing process. Indexing of almost 3000 pages worth of text is considered as sufficient for current indexing needs.

One factor on designing proper search method is to know what kind of data it needs to operate on. By looking at the indexed tsvector contents there appears to be a great deal of indexed terms which are not that easy to use when using keyword search:

```
'-01':38C,39C '00':41C '00.000':42C '04313':7A '2020':37...
```

These exist in the tsvector as the whole metadata JSON entry is indexed. The field might contain date entries which are not that useful on FTS. Same issue exists with enum entries are stored only by their index key as the value is looked later from different table.

```
{"owner": 13, "ownerUnit": 14, "containsConnectionInfo": 1}
```

The “owner 13” is the index key and the actual value for key “13” is read from product type metadata constraint table which list all possible string values an owner metadata field may specify. To increase the precision of results the metadata indexing could be limited to include only the string type fields from the metadata as shown in Listing 14 and change the indexing of enum fields to use the actual metadata value instead of the key.

To execute the queries within the given time constraints a proper use of indexing is required. All the sample queries were run with indexed fields to get optimal performance. The use of indexes will not alter the result of a query, but they provide significant performance benefit when they can be utilized by the database. For downsides, the use of an index adds a small amount of overhead to the queries, and they consume disk space. The PostgreSQL database supports many index types [27, Ch. 11.2] which can be used to speed up the searches. The most important indexes are the Generalized Search Tree (GiST) and General Inverted Index (GIN) as these can be used with the FTS. The database required a helper function given in Listing 17 for indexing fields for the trigram and pattern matching queries.

The details of database query analysis is found in Appendix 7 Section 7.4. The first analyzed query was to see how well the database could return matches for term “kissa” which is cat for Finnish by using various search methods, starting with pattern matching, then with trigram matching and finally with FTS.

Searching the products with simple pattern matching query for “kissa” shown in Listing 18 returned 45 matching records but out of those 39 were true positive matches while 6 were false positives. In the false positives the “kissa” is part of other words like “pankissa” or “at the bank” which has nothing to do with felines. The problem is that the pattern matching does not consider word boundaries, so it returns extra results in addition to correct matches. This could be remedied by adding whitespace in the query term ’ kissa ’ but then it would not match text within words.

The same search query using trigram matching in Listing 19 returned 60 rows as a result containing 21 false positives and 17 false negatives. The fuzzy query is looking for similar words within the fields, so it gives matches for words such as “kassa”. The trigram matching is by default using rule that 60% of trigrams matched will yield a match. This value can be change and after modest increase to 70% the query results were much better. With the changed similarity score the query returned 40 matches with 39 true positives and only one false positive which shows significant improvement.

The sample query gives accurate results when using the FTS with default configuration as shown in Listing 20. All returned rows were correct, but the query failed to return 17 matches which were present when using other matching methods. The default FTS configuration for term “kissa” does not make use of prefix matching which leaves for example compound words starting with “kissa” outside of matches such as “kissatiedote”. Changing the above query to use prefix-matching as in Listing 21 for search terms gave the best result getting perfect precision and recall on this simple query. The Table 13 provides summary of the calculated scores and execution speeds of the various tested search methods. The existing research [6] seems to support that stemming with compound splitting yields similar results as n-gram matching but it notes that stemming has smaller memory requirements.

Table 13: Summary of the PostgreSOL Search Query Results.

Search technique	Results	TP	FP	FN	Precision	Recall	F1	Exec time
ILIKE matching	45	39	6	0	0.867	1	0.929	$\approx$ 5.5ms
TRGM - defaults	60	22	21	17	0.512	0.564	0.537	$\approx$ 35ms
TRGM - with 0.7	40	39	1	0	0.975	1	0.987	$\approx$ 35ms
FTS - defaults	22	22	0	17	1	0.564	0.721	$\approx$ 5ms
FTS - with prefix	39	39	0	0	1	1	1	$\approx$ 5ms

All search methods yielded acceptable results for free-text search using single search term but requirements in Table 9 had other features which will limit the options. The search method must allow multiple search terms, prefix-matching, ranking, phrase search and inflection support as well. Some users of survey listed interest on having the search results highlighted but implementing said feature would require more accurate specification how precisely this should be done, so that requirement is omitted from this analysis.

The pattern matching queries are left out as they will not provide method for ranking of the results nor any inflection support. Inflection support in this context meaning that user does not need to type in the search term accurately for match to occur. This can be implemented either by stemming from FTS or fuzzy matching from pg_trgm. Phonetic matching could be used but these do not scale to longer texts for phrase searches. The above results show that trigram matching with increased word_similarity score gave a slower performance than FTS but similar results. At this point the pattern matching can be dropped as an alternative.

One key difference between the remaining methods is that FTS query can cope with multiple search terms with same kind of performance as single query term. The fuzzy trigram search would need to be repeated multiple times and the results be combined to get same end result.

Indexing multiple fields for the fuzzy search requires more work than using FTS. Another problem is ranking of the results. Where plain pattern matching does not offer any direct method for ranking the results the fuzzy matching has function to return similarity score for matches which can used to implement ranking based on the similarity of matched terms. Adding multiple query terms using weighted zone ranking using trigrams is possible, but it makes the generated queries complicated as illustrated in Listing 22. For comparison the FTS matching version in Listing 23 is easier to understand.

The trigram searching showed good results when working with small amounts of text but the performance of fuzzy matching dramatically drops when trying to search file contents as shown in Listing 24. Another downside on using fuzzy search for file content is that it requires the full contents to be fully stored in the database. Even when proper index the fuzzy search on it would be too slow for any web applications to use. Using simpler pattern matching for file content search is slightly faster than trigram search in Listing 25, but even it does not return responses within the time constraints set in requirements so only search method which fits the requirements is the FTS which copes well even when the text sizes grow significantly as in file content searches shown in Listing 26. To ensure the performance of FTS will not degrade too fast when more products are added into system it was tested with sample data containing one million records of random words in Appendix 7 Section 7.2. The Listing 10 shows that the search is still fast enough even when searching one million records, so the search scales well enough for expected near future growth.

The FTS has plenty of features but it is not without downsides. It does not provide other kind of truncation besides prefix truncation, so it can not be used to match words within other words. This may be remedied by use of lexical analysis. The

PostgreSQL uses the Snowball algorithm for stemming in the default configuration which includes support for all required languages in the new system. The default stemming can be extended by using external contributions such as the dict_voikko to include morphological analysis for Finnish language. The results from various stemming dictionaries in Table 20 in Appendix 7 Section 7.5 for string “Koira- ja kissavakuutushakemus” show that voikko dictionary gives great results as almost all words are in base forms. In addition, it separates compound word “kissavakuutushakemus” into separate words “kissa”, “vakuutus” and “hakemus”. The separation of compound words is the major benefit from lemmatization in this case as with it searching with just term “vakuutus” would give match from above when neither “simple” nor “finnish” configuration would provide a match. This benefit increases if the indexable data has longer compound words. Prior research notes that the “net benefit of compound splitting is usually positive.” [6]

The Appendix 7 Section 7.5 shows comparison results on how the stemming algorithm works on words in sample data in Listing 30. The sample omits the row ID 7 because for some unknown reason it causes the voikko dictionary to crash the database server. This is major issue with the dict_voikko and makes it unusable in production systems until the root cause is identified and fixed, but the project does not seem to be maintained anymore as the latest changes to it were done several years ago.

The PostgreSQL documentation contains sample implementation [27, p. 2624] on how to integrate pg_trgm with FTS by using FTS to generate static word list and checking this smaller list for mistyped words. The project management interviews in Appendix 2 included wish that the search bar would be able to cope with one or two letter mistyping which this would be potential solution. As pointed out by the manual the word list is static and would need to be refreshed from time to time for it to give accurate results.

6.3 Artificial Intelligence Enhancements

The project management and some customers in the survey expressed interest on having AI features in the system as shown in Table 9. These features are commonly used when the amount of data is so large that a manually doing the necessary steps is too much for human operators. Common place where AI is used is to do preprocessing steps for new assets, particularly if the system provides “bulk-loading” i.e. storing numerous assets with one command. The AI could be used to automatically generate metadata tags specific for each asset instead of someone manually typing them in. This kind of feature is commonly called “smart tags” and is available from many DAM vendors.

Other common use of AI is multimedia searches, like searching by images. If given one image, user may ask system to list similar images. These kinds of searches were not listed on the requirements so are not considered in this analysis.

The smart tags are possible to implement, and they may be done before storing the asset into database. This way the choice of data store does not affect the implementation greatly. The other DAM vendors detailed in Appendix 5 mostly used external providers for adding AI features such as Google Vision AI[10] and Amazon Rekognition[11] and the DAM system should initially follow similar path if possible.

Machine translation or rather the more focused Cross-Language Information Retrieval (CLIR) would be good addition to DAM. As DAM stores metadata it might be in any language user is using. One user stores information in Finnish and other user queries this using English ensures these will not match unless some sort of translation is happening. Research indicate that CLIR has almost same accuracy as monolingual IR. [9, p. 448]

[10] https://cloud.google.com/vision
[11] https://aws.amazon.com/rekognition/

6.4 Advanced Search Analysis

Some new systems users wished the system would provide advanced searches commonly available in search engines but not available from relational databases which were discussed in Section 3.5. Some of these features may be added manually using relational databases as well.

Faceted search is a specialized variant of search that allows the user to navigate in a streamlined way with search filters which are presented depending on the searched object. [8, Ch. 9] Many search engines such as Apache Solr provide faceting by default. Relational databases do not directly provide faceting feature but similar effect for simple faceting needs may be achieved by using window and JSON functions as shown in Listing 31 of Appendix 7 Section 7.6.

Autocomplete makes the system give out better search terms for the user while typing the search query. There are few alternatives how to find the “correct terms” from incomplete search input. First is by parsing the system logs for successful search queries, storing the used search terms in the database and using those to suggest better search terms. The suggestion could be done with fuzzy matching so small typos still lead to match and system can use results to autocorrect user query.

Search-as-you-type or predictive search where users search results are narrowed by each key press. Problem in this is to get the searches as lightweight as possible, so they can be executed after each key press without overloading the service. The FTS is only search option fast enough for this task, but it requires custom tokenizer shown in Listing 32.

6.5 Miscellaneous Feature Analysis

The PostgreSQL database already provides search filter features, so there is no need for further work in that regard. The search history for each user is also quite trivial to implement with the current system. The user searches come in the system from two REST-API calls, so the system could be changed to just storing these searches in their own database table as JSON data. The UI can then query this table and re-execute previous searches.

The quick and advanced search feature would need a more thorough specification before evaluating its implementation. The DAM vendor review summary in Table 8 points out that several products had quick and advanced search, but many did not provide accurate documentation on how it was implemented. Google web search provides separate page[12] with advanced filter controls if the search bar was not sufficient and similar design could be implemented in the new DAM as well.

[12]https://www.google.com/advanced_search

7 Discussion and Recommendations

The customer survey search rating suggest that the users are quite satisfied by the current search using FTS and the results seem to support this as users mostly rated keyword search features as a must-have. The current FTS use may be improved with few simple ways to make it achieve better results. First the indexing could be tweaked to be language-specific for each document bringing stemming support. Current system does not have information on how this language is to be selected, but this is a task which may be done by agreeing with the systems users on how to proceed. One method would be to type all information in one language and use that for indexing or use pre-existing content language field to select it, or by having separate field for product/rendition metadata language. With this change the search accuracy can be increase by utilizing stemming and possibly lemmatization.

The Finnish language grammar is hard for stemming algorithms, but it could be improved by using lexical analysis provided by tools such as the Voikko[13]. This would also reduce the need for having pattern matching as Voikko would provide separation of compound words to individual words to help to match single words. The use of Voikko library would require extra effort in identifying and fixing the root cause of the exhibited database crashes while evaluating its use.

For improving the search accuracy indexed fields could be limited to only relevant ones. Currently, the whole customer product metadata field is indexed which index numeric fields which are not easily searched by keywords. They should be either dropped from indexing or the value strings pointed by metadata key fields should be indexed instead. To get file contents in the database require small extension to the current architecture detailed in Figure 7. When storing a new asset file the API would not store the file directly but only register the saving request into internal job queue. The microservice architecture would be extended by having a separate indexer component which would read the job queue and do necessary

[13]https://voikko.puimula.org

preprocessing steps before saving the assets into the data store. This would allow the frontend to resume other work immediately after registering asset saving and long-running asset processing could be done in indexer. First preprocessing step for the indexer would be to parse the file content and index them into the database allowing users do file content searches. Later this step could be extended to cover other tasks as well, perhaps using AI to implement smart tagging etc.

One possible implementation would be to make the current system search as the quick search and implement it by using trigram matching. The quick search would only search product and rendition metadata but not the file contents. For advanced search the search would switch to using FTS and search could be extended to cover file contents as well. Trigram searching would be skipped with advanced search as it does not work well when processing file contents as shown in Listing 24. This two combination of two search methods might be confusing for the clients to use, so it should be tested and implemented carefully.

7.1 Using Search to Fix Known Issues

While the search improvements are important it is essential that the existing issues in it are fixed as well. The following lists possible solutions for the issues listed in Table 3.

To improve the matching process the trigger updating the indexable tsvector should be changed to incorporate above changes as listed in the Appendix 27 which improves the original trigger definition in three ways. Firstly, it tweaks the indexing of username or user's email field to use plain "simple" configuration. This keeps the email as single field instead of trying to split email into separate fields based on some rules. This works in this case as most common email format used by the system is "firstname.lastname@companyName.domainName". The users name is stored separately, and full email is added to index so matching should work with any of them. Keeping email company name or top-level domain do not seem to add additional value as index terms as these are effectively good candidates for stop words. Secondly, it is the using different dictionaries based on the content language of the product row. This makes assumption that the name and description fields are given in same language as the product content language setting. This might not be the case for every customer and needs to be verified when setting up each customer's environment and configured accordingly. Lastly, instead of indexing whole metadata it selects only fields from metadata which contain text data which can be used for keyword matching. The precise list of fields to index varies by each customer. Alternatively the system could look up the indexed field contents and index that instead of the numeric value.

To fix the query parsing a new parser would be needed. Evaluation should be done to see if any existing parser such as pg-tsquery[14] would be enough or should new one be developed for this purpose. The main issue to fix is parsing of quoted text, otherwise the existing solution should be enough. The parser should also handle the special characters such as "ABC 2/2.10" correctly.

The default configuration of the database has inconsistencies when parsing hyphen separated values where the hyphen is sometimes parsed as negative sign for following number as shown in Listing 28. This affects especially most common search using the product codes which consists mostly of hyphen separated numbers and characters. The precise root cause for this is unknown, but the problem can be mitigated by upgrading the database software version and tweaking the FTS indexing options shown in Listing 29.

[14]https://github.com/caub/pg-tsquery

8 Conclusions

The information retrieval or search is seemingly simple thing, but it quickly becomes quite complex topic. There exists multiple ways to achieve searching, each with their own trade-offs. This makes each implementation unique for the given project, a perfect solution for one project might not fit to another project at all.

The aim of this work was to find out problems in current system search, does it provide the wanted features and is the system architecture capable on providing adequate search. The system was using FTS but exhibited some issues which caused users the get unexpected or missing results as pointed in Table 3, but these may be fixed with little effort as detailed in Section 7. The wanted features for the new DAM system were done by examining the new system from several view points collected in Table 4. Each view point gave a new perspective on what the system should provide. Analyzing the view points lead to creation of requirement specification in Table 9 which presents the list of search constraints and features the system should provide along with priority for each. These were summarized in Table 10 into four groups sorted by priority. The results show that keyword-based search is enough for the new system as long as it abides by the constraints given. The next expansion of search features should be adding of AI enhancements into the system. After these the design should focus on providing advanced searches common in search engines and list of smaller miscellaneous features. The analysis done on Chapter 6 shows that the current architecture using FTS search is capable of handling most requirements with adequate performance even when the amount of entries searched is increased. Most difficult part for the new system is expanding it to provide the advanced search features. These may be implemented using current architecture, but it requires considerable effort.

8.1 Future Work

This work was focused on testing the limits of the current tools used by the new DAM system. This leaves a lot of potential good candidates for future research. Primary target would be to examine how a search engine would handle the searches and integrate with the new system. Good candidates for search engines would be the Apache Solr[15] and Elasticsearch[16] as these very often used by other DAM vendors and both of them are based on the Apache Lucene[17] and provide plenty of features. Most of the concerns using them come from increased maintenance and the need to keep the indexed data in sync between the primary data source and search engine. One potential solution to mitigate this would be to evaluate projects such as ZomboDB[18] which allow PostgreSQL database to use Elasticsearch engine transparently from within the database.

Other potential research point would be the use of AI features in DAM. Some users wished to have AI features in the current system and the trend seems to be moving towards using AI more and more it would be best to identify how it could be utilized in the new DAM system as well. There are numerous SaaS options providing AI enhancements giving potential research targets on analyzing how big improvements they could provide in the system. These could be initially used to improve the quality of asset metadata and then be extended in providing more advanced searches such as providing semantic search.

Finally, this work mostly settles on keyword-based search but new customers starting to use the new DAM are already defining more multimedia types in their installations than in previous system. Multimedia files are not directly composed of text making them harder to use with keyword-based search. This area of IR on multimedia content seems to be under active research at the moment, so there would be plenty of opportunities for future work.

[15] https://solr.apache.org/
[16] https://www.elastic.co/elasticsearch/
[17] https://lucene.apache.org/
[18] https://www.zombodb.com/

References

1 Russell, Stuart J. & Norvig, Peter. 2020. Artificial Intelligence - A Modern Approach. Fourth International Edition. Pearson Education.

2 Ng, Andrew. 2018. “Machine Learning Yearning: Technical Strategy of AI Engineers, In the Era of Deep Learning”. Draft. <https://www.deeplearning.ai/wp-content/uploads/2021/01/andrew-ng-machine-learning-yearning.pdf>.

3 Silver, David; Hubert, Thomas, et al. 2018. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: Science 362.6419, pp. 1140–1144. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. <https://www.science.org/doi/abs/10.1126/science.aar6404>.

4 Penttilä, Jeremias. 2015. “Koneoppiminen”. Bachelor’s Thesis. University of Jyväskylä. <http://urn.fi/URN:NBN:fi:jyu-201603211906>.

5 Hagiwara, Masato. 2021. Real-World Natural Language Processing - Practical applications with deep learning. Manning Publishing.

6 Brants, Thorsten. 2003. “Natural Language Processing in Information Retrieval.” In: CLIN 111.

7 Christopher D. Manning Prabhakar Raghavan, Hinrich Schütze. 2009. An Introduction to Information Retrieval. Online. Cambridge University Press. <https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf>.

8 Gupta, Anuj; Majumder, Bodhisattwa, et al. 2020. Practical Natural Language Processing. 1st Edition. O’Reilly Media, Inc.

9 Baeza-Yates, Ricardo. 2004. “Challenges in the interaction of information retrieval and natural language processing”. In: *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, pp. 445–456.

10 Guarino, Nicola. Jan. 1995. “Ontologies and knowledge bases: towards a terminological clarification”. In: pp. 25–32.

11 Shahi, Dikshant. 2015. Apache Solr: A Practical Approach to Enterprise Search. 1st Edition. Apress.

12 Van Rijsbergen, C.J. & Van Rijsbergen, C.J.K. 1979. Information Retrieval. 2nd Edition. Butterworths.

13 Gusfield, D.; Press, Cambridge University & Fund, John D. & Phyllis S. Harrah Library. 1997. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. EBL-Schweitzer. Cambridge University Press.

14 Bates, Marcia J. 2011. Understanding Information Retrieval Systems. 1st Edition. Auerbach Publishers.

15 Heini, Jari-Pekka. 2010. “Organisaation tiedonhaku - tarkastelussa avoimen lähdekoodin ratkaisut”. MA thesis. University of Jyväskylä. <http://urn.fi/URN:NBN:fi:jyu-201008272500>.

16 Valkonen, Juho. 2015. “Document management for small business”. MA thesis. Turku University of Applied Sciences. <http://urn.fi/URN:NBN:fi:amk-2015092814956>.

17 Paananen, Anna. 2012. “Comparative Analysis of Yandex and Google Search Engines”. MA thesis. Metropolia University of Applied Sciences. <http://urn.fi/URN:NBN:fi:amk-2015092814956>.

18 Haynes, David. 2017. Metadata for Information Management and Retrieval. Ed. by Haynes, David. 2nd. Facet Publishing.

19 Spärck Jones, K.; Walker, S. & Robertson, S.E. Aug. 1998. A probabilistic model of information and retrieval: development and status. Tech. rep. UCAM-CL-TR-446. University of Cambridge, Computer Laboratory. <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-446.ps.gz>.

20 Robertson, S.E. & Spärck Jones, K. Dec. 1994. Simple, proven approaches to text retrieval. Tech. rep. UCAM-CL-TR-356. University of Cambridge, Computer Laboratory. <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-356.pdf>.

21 Sumathi, S. 2007. Fundamentals of Relational Database Management Systems. Ed. by Esakkirajan, S. Springer Berlin Heidelberg.

22 Austerberry, David. 2012. Digital Asset Management. 2nd Edition. Taylor & Francis.

23 Saareks, Jani. 2019. “Tehokkaan tekstihaun toteuttaminen käyttöoikeudet huomioiden”. MA thesis. University of Jyväskylä. <http://urn.fi/URN:NBN:fi:jyu-201906073064>.

24 Latif, Afshan; Rasheed, Aqsa, et al. 2019. “Content-Based Image Retrieval and Feature Extraction: A Comprehensive Review”. In: Mathematical Problems in Engineering 2019. Ed. by Lefik, Marek, p. 21.

25 Keathley, Elizabeth Ferguson & Gyor, Henril De. 2014. Digital Asset Management: Content Architectures, Project Management, and Creating Order out of Media Chaos. 1st Edition. Apress.

26 Leppänen, Sandra. 2017. “Leveraging Digital Asset Management (DAM) in a Finnish retail corporation”. MA thesis. Yrkeshögskolan Arcada. <http://urn.fi/URN:NBN:fi:amk-201704104499>.

27 Group, The PostgreSQL Global Development. 2021. PostgreSQL 12 Documentation. ver. 6. Accessed: 2021-06-27. The PostgreSQL Global Development Group. <https://www.postgresql.org/files/documentation/pdf/12/postgresql-12-A4.pdf>.

28 Göker, A. & Davies, J. 2009. Information Retrieval: Searching in the 21st Century. 1st Edition. Wiley.

29 Levene, Mark. 2011. An Introduction to Search Engines and Web Navigation. 2nd ed. Wiley.

30 Google. 2020. Trillions of Questions, No Easy Answers: A (home) movie about how Google Search works. <https://www.youtube.com/watch?v=tFq6Q_muwG0>. Visited on 01/16/2022.

31 Colin Reid, Mike McGuire. Dec. 2020. Market Guide for Digital Asset Management. Tech. rep. Accessed: 2021-02-04. Gartner. <https://www.nuxeo.com/resources/gartner-market-guide-dam>.

32 Barber, Nick. Nov. 2019. The Forrester Wave: Digital Asset Management For Customer Experience, Q4 2019. Tech. rep. Accessed: 2021-02-06. Forrester. <https://reprints.forrester.com/#/assets/2/376/RES146956/reports>.

33 Bartunov, Oleg & Sigaev, Teodor. 2007. Full-Text Search in PostgreSQL: A Gentle Indroduction. Tech. rep. Moscow University. <https://www.pgcon.org/2007/schedule/attachments/12-fts.pdf>.

34 Hellerstein, Joseph M & Pfeffer, Avi. 1994. The RD-tree: An index structure for sets. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences.

35 Helmer, Sven. 1997. “Index structures for databases containing data items with set-valued attributes”. In: Technical Reports 97.

36 Korenius, Tuomo; Laurikkala, Jorma; Järvelin, Kalervo & Juhola, Martti. Jan. 2004. “Stemming and lemmatization in the clustering of Finnish text documents”. In: *Proceedings of the thirteenth ACM international conference on Information and knowledge management, CIKM’04*, pp. 625–633.

37 Voikko. 2021. Voikko - General architecture. Internet. Accessed: 2021-08-14. <https://voikko.puimula.org/architecture.html>.

1 Interviews

This appendix lists the questions asked in project member interviews. The aim of the interview was to get overview on how the search should function. Each interview was done one on one, so we get individual views on the issue.

1.1 Information Retrieval Design

Term: a single search term used in search

The interview used simple priority system called MoSCoW to set priorities for features:

1. Must
2. Should
3. Could
4. Wish

How many years of experience in using DAM systems?

How much data and what kind of data should be search cover?

What kind of material should be searched?

How much of data should the system handle?

How fast the system should yield search response?

How fast the system should index the data?

How long delays are permitted into indexing, should it be real-time, near real-time, or can it be delayed further to every 5 minutes etc.

What is the architectural scope for the improvements?

The question had choices to scope how large changes can be made into the system. The options were:

- Search features must use current architecture and tools
- Search features must use current architecture and can extend it with new tools
- The current architecture can be totally redesigned

How large budget is given for the search improvements?

Idea is to scope out how much money can be used in the search feature, can it use commercial tools or stick with free solutions like open source software. Also the idea is not to get accurate limits, just general theme of things like is it free tools only, tens of euros per month, hundreds euros per month and so forth

Can the search use cloud service providers?

Should the search bar be customizable to each customer?

How quickly the search improvements should be placed into production?

Idea is the gauge once the search improvement is presented, how large project it can be, can the process take days, months or even years.

What the search should yield as a result, products? renditions?

What fields should be indexed for searching?

Should the search cover file contents as well?

Should the search be implemented just as a search bar?

Should the search bar be separated as Quick search and Advanced search?

UI design suggestions?

Where the search term should match?

Should the search term match word prefix, suffix or anywhere?

Should the search support multiple search terms?

If multiple terms, How to combine them by default, AND?

If multiple terms, What operations should be supported? AND, OR, NOT?

Operator precedence, should parenthesis be supported as well?

Should the search support phrase search?

Should the search be exact or fuzzy?

If fuzzy, any preference how to implement it?

Levenshtein, Soundex, trigram

Should the search work on multiple languages, which ones?

If using fuzzy search, what features should it cover?

- Stemming?
- Lemmatization?
- Compound word handling?
- Stop words?
- Synonyms?

Should the system support field-restricted search queries?

For example queries such as: `code:F200 AND auto` So that string “F200” is only matched in code field and word “auto” in any field.

Should the search provide auto-complete?

Should the search provide suggestion?

Should the search provide spellchecking?

Should the search provide correction?

Should the search provide search-as-you-type?

Should the search results be ranked based on relevance?

Should the search highlight where the match was found in the results?

Should the search support machine translation?

For example if keywords are indexed in English but user types in Finnish keyword, should it automatically translate the keywords into English.

Should the search implement image recognition?

When giving keyword it tries to match it based on where it appears in the images.

Should the search be able to seek within the audio or video content?

Giving search term, should it try to match it from spoken words in audio or visuals from video feed.

Should the system support stored searches?

If stored searches, should they be organization-wide or personal?

Can the system default searches be customizable?

Can the user for example set some default search when going to product page.

2 Interview Results

This appendix presents the results of the project management interviews done as part of the thesis.

The interviewees agreed in that the new system must cope with managing assets for the current small-scale customer deployments, but it should be easily scaled for bigger deployments. Initial minimum requirements for the search feature is to be able to handle tens of thousands assets, some rough guidelines on asset file sizes were tens of megabytes for images, hundreds of megabytes for documents and gigabytes and larger for video files. The priority between these are that text document support must be done first and foremost followed by image support. The current platform does not have much video support yet, but it was seen as essential feature to have in near future. The search feature should be fast, it should yield responses within a second or two at maximum. If the system is searching through file contents as well, it can use few more seconds to come up with answer.

The interviewees agreed on that the current system architecture may not be replaced entirely as the budget does not allow for full system redesign at this point. The current architecture may be extended with new tools, provided that they work together with the current design. The tools used may include proprietary solutions as long as those are not too expensive to use. The implementation for the improvements may not take too long, they should be finished in months or at maximum within a year.

One of the mandatory features to have in the system is the indexing of file contents and metadata. The priorities in file content indexing is that the text content indexing must be done first, followed by the indexing of file metadata. Once those are done the system can be extended to analyze file contents more thoroughly such as using AI to recognize concepts from images. And finally focus on video and audio by analyzing the content for language, speech-to-text and such features if a feasible method is found to achieve those. These features would allow the limit search to return all video files with spoken Finnish talking about car sales and similar searches.

The interviewees did not agree on what would be exactly the scope of the search but were unanimous that all product information must be indexed for the search. In the DAM system a product can have multiple renditions. These renditions represent a single physical file and its metadata. Interviewees did not agree should the search also query rendition data as well as product data by default. They saw it might make the system too slow to use if the search scope would be set to be too large.

Interviewees wanted the search query to match patterns on any part of the document words and if that would not be possible at least use prefix-matching. Search bar should also support multiple search terms which would be joined together with boolean AND condition. Interviewees did not see immediate need for rest of boolean search options such as “OR” or “NOT”, neither for ability to setting search term precedence with parenthesis. Phrase search was required once file contents would be indexed. The search feature should support Finnish, English and Swedish languages as those are the main languages used in the market area. Ranking the search results by relevance was one required feature in the interview.

Having field-restricted search was not seen as an important feature by the interviewees. This feature would limit the searching to a single field of the document, similar to Google web search `site:metropolia.fi courses` were system would match “courses” search term only on metropolia.fi domain. Same feature in DAM context could be to limit the search to only product names, product codes or some other field. This was considered to be confusing for the systems users and feature might not be useful to have.

The auto-complete, correction, suggestion and search-as-you-type kind of features were not seen as necessary at this point. They could be useful features to have but research and development focus should be on getting the more critical features implemented before looking at these.

Using AI to enhance the search was not seen as first priority but something to be added into the system later on. One of the immediate need for AI enhancements would be indexing of image file contents by using image recognition. This would allow users to search by image concepts.

And lastly, it was seen useful to store the user’s search history. It would seem useful to keep record of, for example the last five search queries, but this was not seen as something of a high priority. Saving of user search queries or changing the system default searches was not seen as critical at this point.

The interview provided good framework on how the search should work and what areas require more thorough analysis. The combining of search terms with boolean terms and fast response times seems to indicate that the full text search would be a good fit. Problem with full text search is that it does not allow pattern matching besides prefix-matching with PostgreSQL. Part of interviewees expressed their wish that search bar would allow user to make one or two typing errors in search term and the system would still return the wanted result. Something like when typing product code but miss-typing single letter would still return some results including the intended product code by using similarity search.

3 Customer survey

This appendix lists the questions asked in the customer survey. The aim of the survey was to get gather quick overview on how good users of the system feel the search is and to rate which features are most important to have in their workflows.

The survey is short and quick to fill so more users would incline to answer to it. Main idea is to gather user rating before starting to rework the system and redo the survey once the features are ready and users have had time to use them to see if the user view of system improves.

3.1 Core Survey - Search

Core search feature customer survey is meant to gather info about the current state of search in Core. It also tries to identify most important search features.

The survey is split into several sections, of which 5 includes the questions. The survey is done anonymously unless participant chooses to give consent for interview. The consent is asked at the end of the survey.

It takes approximately 15 to 20 minutes to fill out this survey.

Rate current search features in Core

On scale of 1 to 10, how would you rate the current search implementation. Higher scores the better.

3.1.1 Basic Search Features

This section asks you to rate, how important the basic search features are to your work. The scale of answers go from 1 to 5, so that 1 = not at all important, 3 = can’t say, 5 = mandatory.

Truncate search

Truncate search means the ability to truncate the search term at given point to increase its accuracy. For example “form*” would return all documents were exists word starting with string “form”.

Field-specific search

Field-specific search means ability to restrict text search to specific fields. For example typing “code:200L” would return all documents where code-field would contain string “200L”.

Search filters

Search filters mean combo boxes, which allow to filter search to specific language, product group etc.

3.1.2 Free Text Search

This section asks you to rate, how important various text search features are to your work. The scale of answers go from 1 to 5, so that 1 = not at all important, 3 = can’t say, 5 = mandatory.

Free text search

Free text search means search bar which allows free form text to be typed which is used to search documents.

Support multiple search terms

The system search bar should allow to give multiple search terms and allow to combine them with operators AND, OR and NOT. For example search query “form AND cat AND NOT dog” would return all documents which contain words “form” and “cat” but don’t have word “dog” in them.

Phrase search

Phrase search means ability to search longer text phrases. For example “Mat’s car dealership” would return documents which contain exactly “Mat’s” followed by “car” followed by “dealership”. Search would exclude documents were only one term term matches or where all terms exist but are not in same arrangement.

Proximity search

Proximity search means ability to limit search by word proximity. It allows queries like: return all hits were word “dog” exists within 5 words of word “hotel”.

Word inflection support in search

Should the search know how to match search terms all word inflections automatically? For example when searching with term “dog” the system would return all documents where there exists any inflection of word “dog” like “dog”, “dogs” etc.?

Result ranking

The system should rank the results by where match was found. For example search results where query term was matched in name should be ranked higher than those where match was in description text.

Match highlightning

The search results should highlight were match was found?

Similarity search

Search should also return matches on similar search terms, for example when searching term “1004N” it should give results also for terms “2004N”, “1003N” etc. allowing few typing errors in search query.

3.1.3 Advanced Search

This section asks you to rate, how important various advanced search features are to your work. The scale of answers go from 1 to 5, so that 1 = not at all important, 3 = can’t say, 5 = mandatory.

Autocorrect

The system should automatically offer to correct mistyped terms while typing the search query?

Search autocomplete

System should offer to autocomplete search terms while typing them?

Search suggestions

System should automatically suggest the next search term while typing the query?

Did-you-mean search

The system should offer better search terms when showing search results if those would return more results?

Search-as-you-type

The system should run the search automatically after each key press?

Dynamic search facets

After showing results of query the system should offer dynamically generated facets to limit search results. Once results are filtered any filter component not able to filter results further gets removed so only relevant controls remain.

Smart tags

System should add tags to products and renditions automatically using AI?

Machine translation in search

System should use automatic machine translation to convert asset metadata into unified language and translate the search query to matching language.

3.1.4 Scope of Search

This section asks you to rate, how important given search scope is to your work. The scale of answers go from 1 to 5, so that 1 = not at all important, 3 = can’t say, 5 = mandatory.

Include version information to search

The system should include product version information in addition to product information when searching?

Search should contain version file contents in search
Search should also try to find matches from version file text contents?

Search should include file's metadata Search should include version file's own metadata as well as system provided metadata in search?

Visual search
The system should include image and video files in text search? For example when searching term "cat", a pictures of cats or videos of cats should also be returned as matches

Audio search
The system should search also seek within audio files and video soundtrack for matching speech. Text search should return match if any file's speech matches the search term.

Content similarity search?
The system search should be able to provide similar file results? For example from image product there would be option to list similar images?

Location search
System search should be able to search by location information? When searching term "paris" it should return all images taken in Paris etc. Or the version page would allow to search for files from same location.

3.1.5 Personalisation of Search

This section asks you to rate, how important given search personalization is to your work. The scale of answers go from 1 to 5, so that 1 = not at all important, 3 = can't say, 5 = mandatory.

Saved searches, personal
User should be able to save searches for later execution?

Saved searches, organisation-wide

User should be able to save searches for later execution and share them to others in organisation?

Custom default searches

User should be able customize the default search run when going for example products page, it would show only products owned by user etc.

Organisation-wide synonyms

Organisation users should be able to add synonyms to search terms to improve accuracy. For example “tv”, “telly” would be stored as “television” internally.

Quick and Advanced search

Current search controls should be separated to Quick search (just search bar) and advanced search (adds filter boxes, checkbox to include file contents to search etc.)?

3.1.6 Conclusion

Thank you for your answers so far. We still have few open optional questions left before you can send the answers.

Open feedback about the survey?

Consent for interview?

If you’re willing to participate in a about hour long interview based on the answers given so far, type in your email address in the field below. Participation is not mandatory and not all volunteers will be interviewed. You can leave the field empty if you don’t wish to participate in the interview.

$$\begin{aligned}\overline{x} &= \frac{(8+6+5+8+8+3+8+2+9+8+8+8+5)}{11}\\ &= 7.818182\end{aligned}$$

Listing 2: Calculation of Average for Customer Survey Search Ratings.

4 Customer Survey Results

This appendix contains the customer survey results. The survey was implemented as a Google Forms survey and was sent to 56 recipients. The users were given three weeks to answer the questions and survey received total of 13 responses during that time. The responses appeared to contain valid input except for two answers which stood out from the others. In both answers the core system overall rating was given as average to low and all the search feature questions were answered by score of 1, not at all important. Finally, in the open feedback the user had given almost same response in the both answers by saying paraphrasing "I have not been able to log in the system" which would indicate that both answers came from the same person. The first question was to rate the system search features which require user to be able to log into the system, so these two responses seem invalid. This study is done without those answers. With remaining answers the survey response percentage is 20% overall, which gives enough results for this analysis.

After the survey had been completed, it was noticed that the answer option for the "Field-specific search" question was not present in the questionnaire, so it does not have any answers.

The first question in the survey was to rate how well new systems search functionality is working, and the average score is presented in Listing 2. The users of the system seemed to value the current features already quite high with average score of 7.8. This would indicate the users generally like the features, but there is still room for improvement.

Table 14: The Scores Given in Customer Survey.

Feature	Ratings										
Truncate	3	5	4	4	5	5	3	5	5	5	3
Filters	2	4	4	2	4	4	5	4	1	5	4
Free-text	2	5	4	4	5	5	5	5	3	5	4
Multiple terms	3	1	3	3	1	4	4	5	3	1	3
Phrase	1	1	1	3	4	4	4	2	3	1	2
Proximity	1	1	1	1	1	2	3	3	4	1	2
Inflection	2	5	4	3	4	4	4	5	3	3	3
Ranking	4	4	2	4	5	5	4	4	5	3	3
Highlighting	3	2	4	4	4	5	3	4	4	1	4
Similarity	2	2	2	3	2	3	1	2	1	4	3
Autocorrect	2	1	1	3	2	5	2	4	1	4	4
Autocomplete	2	2	1	3	1	4	1	4	1	1	4
Suggestions	2	4	2	3	1	4	3	3	1	4	4
Did-you-mean	2	2	2	2	2	4	2	3	1	2	4
Search-as-you-type	2	2	4	2	2	4	1	4	3	5	4
Facets	2	2	2	3	3	3	1	4	1	4	4
Smart tags	2	4	2	4	4	4	3	4	1	4	4
Machine translations	2	1	2	3	2	3	4	4	4	1	4
Version info	2	2	2	3	3	3	3	4	1	1	4
File contents	2	3	2	3	2	3	2	4	1	1	4
File metadata	3	4	5	4	1	2	2	3	1	1	3
Visual	1	2	4	2	1	3	1	3	1	1	2
Audio	2	2	4	2	2	3	1	3	1	1	3
Content similarity	2	2	2	2	2	2	3	3	2	1	3
Location	2	2	1	2	1	3	1	4	1	1	3
Personal Saved	2	4	3	4	3	3	1	4	3	1	3
Org-wide saved	1	1	3	3	2	3	1	4	1	1	3
Custom searches	2	2	3	4	3	3	4	4	3	1	3
Synonyms	2	2	1	2	3	3	1	4	1	1	3
Quick/Advanced	3	4	3	4	1	3	1	4	1	4	4

To analyze which features users want to use in a DAM the answers were scored using two methods with scores detailed in Table 15. The first method is looking at how majority thought about features, giving each feature either positive or negative score based on the rating and summing the total of all scores. The another method used was to ignore negative responses and only focus on the positive responses. This makes the assumption that addition of the feature that is not wanted by all would not affect negatively on all users.

Table 15: Scoring Methods for Customer Survey Results.

Score	First method (total)	Second method (positive)
5	+2	+2
4	+1	+1
3	0	0
2	-1	0
1	-2	0

For analysis sum of scores was calculated for each answer, both from all answers and filtered answers where the two responses described above have been dropped. The results of scores are collected in Table 16. Features which had only negative scores are not listed here as those are not interesting for this study.

The sum of answer scores are then transformed into MoSCoW priorities where tasks are put into one of following categories: must-have, should-have, could-have, and will-not-have. This analysis uses following scale for assigning the priorities:

Table 16: The Scoring of Search Features from Customer Survey.

Feature	All scores	Only positive scores
Truncate search	14	14
Search filters	6	10
Free text search	14	15
Support multiple terms	-2	4
Phrase search	-7	3
Proximity search	-13	1
Word inflection support	7	8
Result ranking	10	11
Match highlighting	5	8
Similarity search	-8	1
Autocorrect	-4	5
Search autocomplete	-9	3
Search suggestions	-2	4
Did-you-mean search	-7	2
Search-as-you-type	0	6
Dynamic search facets	-4	3
Smart tags	3	7
Machine translation in search	-3	4
Include version information in search	-5	2
Search should include file contents	-6	2
Search should include file metadata	-4	4
Visual search	-12	1
Audio search	-9	1
Content similarity search	-9	0
Location search	-12	1
Saved searches, personal	-2	3
Saved searches, organization-wide	-10	1
Custom default searches	-1	3
Organization-specific synonyms	-10	1
Quick and advanced search	-1	5

Table 17: Customer Survey Priority Assignments.

Priority	Score
Must-have	Score of 10 or above
Should-have	Score of 5 or above
Could-have	Score of 0 or above
Will-not-have	Scores less than 0

5 Digital Asset Management Vendors

The ten leading DAM vendor products were reviewed as part of this thesis. This appendix lists each tool in bit more detail starting from challengers and moving up.

The Northplains Telescope[19] was listed as the only challenger has since been re-branded[20] as a modular Northplains NEXT platform in which the Xinet seems to provide the DAM features. The Northplains review is based on the Xinet documentation[21]. The Xinet seems to provide pretty good coverage on keyword search options. This seems to be due to possible integration with Apache Solr. The administrator of the system may choose which search engine to use with the Xinet, either full-text search or external Apache Solr. The features of Xinet search include either quick or use more advanced search filter using keywords or free text search. The search handles multiple terms, offers phrase search with proximity matching, restricting search to specific field and is capable of showing the results with dynamic facets. Complex searches may be saved for further execution.

The next DAM vendor group was the categorized as the contenders which included two solutions, Canto and Digizuite DAM.

The Canto DAM[22] appears to be focused on marketing material handling. It provides the basic search features such as keyword search, field-restricted search, supports multiple search terms by boolean operators. The search is capable of looking for matches in the metadata and also from the body content of the asset. It also allows users to save previous searches for easier re-execution. It has integrations to Amazon Rekognition AI services which was used for example in providing automatic tagging of assets. It for example provides facial recognition which allows users later find assets with happy faces in them.

[19] https://www.northplains.com/
[20] https://www.northplains.com/content-lifecycle-management-clm-101
[21] http://docs.xinet.com/docs/Xinet/19.2.1/AllGuides/
[22] https://www.canto.com/

The Digizuite DAM[23] had more options available than the Canto, in above the traditional free-text search it had Apache Solr integration for adding more options for configuring the search functionality. The administrator is able to choose between different styles of search matching, either inflectional, thesaurus or exact. The version 5.5 added machine translation of metadata and provided searching of asset content. The content search was implemented using Optical Character Recognition (OCR) provided by Microsoft Azure Cognitive Services. This kind of content search was available for PDF, JPEG, PNG, BMP, TIFF file formats.

As above lists the contender category products already have quite extensive feature set regarding search. Following contenders were the strong performers which had multiple vendor solutions listed.

The Widen Collective[24] was listed having two kinds of searches, a quick and file content search. The quick search was for searching the asset metadata and gave option to do exact, phrase, or field-specific searches, but it did not support wildcard matching. The file content searching was supported for Office and PDF documents using FTS which supported stemming and substring matching. The search also provides predictive search which can be called a search-as-you-type.

The Nuxeo Platform[25] had quite extensive set of search features. It makes use of multiple backends for data, a RDBMS solution is used as the primary data store, and it is linked with an Elasticsearch search engine for providing advanced asset search capabilities. The Nuxeo supports all basic features such as search truncation and filters. It provides full-text search with multiple terms with stemming, phrase searches, offers suggestions and the search-as-you-type feature. The platform also offered to save searches for later use and to share these with other users. It also allows searching deleted assets from thrash.

CELUM Content Collaboration Cloud[26] had very little publicly available information

[23] https://www.digizuite.com/digital-asset-management
[24] https://www.widen.com/solutions/digital-asset-management
[25] https://www.nuxeo.com/solutions/dam-digital-asset-management/
[26] https://www.celum.com/en/digital-asset-management-software/

about the platforms search capabilities. Analyzing the latest release notes[27] along the marketing info gave some information about the systems capabilities. CELUM provides full-text searching with ranking, and it is possible to do phrase searches as well. Search is using current page context to search, so if user is browsing within a category and does quick search it will only search for matches from within the category but provides option to expand the search to cover all assets if needed. More search engine type feature is the search-as-you-type, and it uses some AI model in providing smart tags when saving assets.

The final member of strong contender group is Bynder Flagship[28] which has quite an extensive list of search options as it uses Apache Solr as its search engine. It provides full-text searching with relevancy ranking and phrase searches. In addition to these it allows field-restricted searches and stores search history so users can rerun recent searches. The system also indexes the file contents for Office and PDF files and offers match highlighting. When working on the UI it has “sticky searches” were consecutive search queries filter the previous search results further allowing users to zone in the assets. For providing search-as-you-type queries the system uses egde n-grams on select fields where it indexes the first 20 characters so that word such as “House” is indexed as a following set: `{'h', 'ho','hou', 'hous', 'house'}`. As Bynder is hosted on Amazon platform it integrates with the Amazon Rekognition to provide AI features which are used for example on providing smart tagging of assets.

The final group of DAM vendors were the market leaders. These are the largest and most featureful vendor offerings with business revenue in hundreds of millions or over. The market research identified three operators in this group.

First market leader is OpenText Media Management[29]. First thing from OpenText is the vast ecosystem it has, it is not just asset management, but it integrates to other OpenText offerings to provide plethora of features. This makes analyzing

[27] https://www.celum.com/en/blog/celum-contenthub-21-9-release-whats-new/
[28] https://www.bynder.com/en/products/digital-asset-management/
[29] https://www.opentext.com/products-and-solutions/products/customer-experience-management/digital-asset-management/opentext-media-management

what features the platform provides a difficult task as the features are provided by different components which might be interconnected or not. For example the OpenText Media Management solely provides the DAM features but can it be integrated with OpenText Magellan Text Mining for adding ML models to providing sentiment analysis and language detection etc. features. This is not covered in the materials. For certain the features in Media Management component are keyword searching using multiple terms matching asset file metadata and contents. Results can be filtered by dynamic facets. For more exotic search features it provides AI search by using the Google Vision AI[30] and Microsoft Azure Computer Vision[31]. The search allows searches for assets based on the properties of images, number of people, facial expression, age, gender, description of image content, objects in the image or colors. These searches can be executed on video as well to find where the AI is used to make speech-to-text transcripts, it also tags known celebrities identified in the video. OCR techniques are used to parse text from images and identify known brands and label in them.

Next listed leader was the Aprimo DAM[32]. The basic search features present in Aprimo did not provide anything noteworthy compared to other vendors. It supports truncate search and filtering. Search terms may be combined with boolean operators and full-text search can be used in select cases and the results may be ranked by relevancy and filtered by dynamic facets. The product also has very extensive API for making searches.

In addition to these there are multiple methods where AI is used to enhance the searching. These include the common way of automatically tagging assets but Aprimo extends this by offering an option to train a business-specific ML model tailor the tagging process for each customer. The search in Aprimo can use AI models to index speech on video and visual texts from video sources. Besides these the search can be used to find visually similar content.

[30] https://cloud.google.com/vision
[31] https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/
[32] https://www.aprimo.com/platform/digital-asset-management/

The final vendor was Adobe with their Experience Manager Assets[33] which is quite large system. Technical side the system is built with Java on Apache Jackrabbit Oak project. The default search features of Oak are used by default which can be enhanced by using Apache Solr integration. The system basic search capabilities are pretty much those of Apache Solr so truncate search with filtering, full-text search with multiple terms and ranking, phrase and similarity searches, suggestion of search terms. Finding similar content or images.

Adobe allows searching their own Adobe stock photo archive to enhance existing assets or make new assets based on the stock photos. Smart collections where users may create new groups from search results. The metadata of assets are translated automatically by AI. The Adobe Sensei AI is also used to add smart tags to the assets. It also makes possible on searching audio and video content. The smart tags feature can also use Smart content service to train a customer specific ML model to applying tags to assets.

Looking through each category of DAM products gives us common themes in them. The challenger-level product provides pretty default search options based on dedicated search engine. At contender group products the basic search engine uses are better integrated and offer more features. With these vendors there starts to be little utilization of AI in aiding in asset searching. Mostly in automatically tagging assets at creation of time.

The strong performers of DAM vendors did not really add anything special compared to that of contenders. They offered more of the same features which were given on the contender-level. The AI features were clearly using external systems like Amazon Rekognition and Google Vision instead of offering any in-house AI solutions.

The market leaders were very distinct group were two things are featured prominently. One is that the products are big and modular. Instead of getting single tool you get a platform you can extend to fit your needs. This makes

[33] https://business.adobe.com/products/experience-manager/adobe-experience-manager.html

identifying all the possible features difficult just by looking at their documentation and materials as they do not make clear which parts can be integrated together and how well that actually works. The other identifying aspect from the leader group is the use of AI. It was used extensively throughout the vendor platform and not just for searching. Possibility that these bigger vendors can hire dedicated professionals to create and train the AI models for each task.

Examples of search features using AI in their tools include:

Smart tagging Automatically identify characteristics, visual or others and create metadata tags. May contain business-specific options if AI can be trained for each customer. Usually offered as part of bulk-loading of assets

Visually similar searches Find images with similar content, Images with same model, happy images etc.

Dynamic facets Create facets automatically based on search results while using AI to filter unnecessary facets

Multilingual search Allow search to use any language but use machine translate to convert queries into target language

Search on Speech in Video search fragments of speech to find matching videos

Search on Visual Text in Video search visual text in video such as from credits or written texts shown on feed

Sentiment analysis Identify and highlight what people are saying, for example social media posts about asset, are they positive or negative

Text mining, natural language processing and understanding Extract parts of text and derive understanding of emotions or intent. The Computer Vision (CV) may be used to discover prohibited materials such as use of alcohol, drugs and violence or adult content.

Document digitization Automate classification and entity extraction with aid of ML. The captured information is further directed to workflows for context-based processing.

Above lists only some samples of search features where AI was used, there were many other areas where AI was also utilized. The material left impression that these use in-house developed AI solutions but looking only at product marketing and internal documentation can not verify this.

A summary of the features present in these vendor systems are given in Table 18. The score represents how many DAM products had the said feature, so scores are given from 0 (none) to 10 (all of them).

Table 18: Summary of Provided DAM Features of Top 10 Vendor Products.

Feature	Score
Truncate search	7
Search filters	9
Free text search	9
Support multiple terms	9
Phrase search	7
Proximity search	3
Word inflection support	5
Result ranking	5
Match highlighting	2
Similarity search	1
Autocorrect	1
Search autocomplete	1
Search suggestions	3
Did-you-mean search	1
Search-as-you-type	4
Dynamic search facets	6
Smart tags	7
Machine translation in search	2
Include version information in search	0
Search should include file contents	5
Search should include file metadata	2
Visual search	4
Audio search	2
Content similarity search	2
Location search	0
Saved searches, personal	3
Saved searches, organization-wide	1
Custom default searches	0
Organization-specific synonyms	1
Quick and advanced search	5

6 File Metadata Analysis

This appendix provides a more detailed look on the file metadata relating to the new DAM system.

Apache Tika[34] is Java-based tool to parse files for their metadata and content. The metadata output varies by file types. Here are some examples of file metadata from different file types. The metadata was extracted with command `java -jar tika-app-1.26.jar -m -j FILEPATH | jq`. The *jq* command is used to pretty print the JSON output given by Tika tool. Simple text files such as JSON files do not contain much metadata in them as detailed in Listing 3.

```
{
  "Content-Encoding": "ISO-8859-1",
  "Content-Length": "4137",
  "Content-Type": "application/json; charset=ISO-8859-1",
  "X-Parsed-By": [
    "org.apache.tika.parser.DefaultParser",
    "org.apache.tika.parser.csv.TextAndCSVParser"
  ],
  "resourceName": "zmg.json"
}
```

Listing 3: Sample of a JSON File Metadata.

The is vast difference in amount of metadata once looking at more complex file such a PDF document. The Listing 4 presents the sample metadata from a PDF presentation. There exists many useful fields in it which could be used to enhance search capabilities of the DAM. It also provides several fields which are of no direct use such as *pdf:charsPerPage* field. When storing the metadata into the DAM system it would not be most efficient to try to index all of the PDF metadata, but to pick the most relevant fields for indexing.

After PDF files the most common file types in the system are the Office files as shown in Table 11. A sample from a Microsoft Word document is shown in Listing 5 illustrates how the metadata is more limited than the metadata on PDF.

[34] https://tika.apache.org/

One noticeable aspect is that many fields are duplicated with different names, such as *Paragraph-Count* and *meta:paragraph-count*. This might be done to keep metadata backwards compatible with older versions.

```
{
  "Author": "Joe Conway joe.conway@crunchydata.com mail@joeconway.com ",
  "Content-Length": "244110",
  "Content-Type": "application/pdf",
  "Creation-Date": "2015-10-28T13:55:15Z",
  "Keywords": "",
  "Last-Modified": "2015-10-28T13:55:15Z",
  "Last-Save-Date": "2015-10-28T13:55:15Z",
  "PTEX.Fullbanner": "This is pdfTeX, Version 3.1415926-2.5-1.40.14 (TeX Live 2013/Debian) kpathsea version 6.1.1",
  "X-Parsed-By": [
    "org.apache.tika.parser.DefaultParser",
    "org.apache.tika.parser.pdf.PDFParser"
  ],
  "access_permission:assemble_document": "true",
  "access_permission:can_modify": "true",
  "access_permission:can_print": "true",
  "access_permission:can_print_degraded": "true",
  "access_permission:extract_content": "true",
  "access_permission:extract_for_accessibility": "true",
  "access_permission:fill_in_form": "true",
  "access_permission:modify_annotations": "true",
  "cp:subject": "Text Search and Pattern Matching",
  "created": "2015-10-28T13:55:15Z",
  "creator": "Joe Conway joe.conway@crunchydata.com mail@joeconway.com ",
  "date": "2015-10-28T13:55:15Z",
  "dc:creator": "Joe Conway joe.conway@crunchydata.com mail@joeconway.com ",
  "dc:format": "application/pdf; version=1.5",
  "dc:subject": "",
  "dc:title": "Where's Waldo? - Text Search and Pattern Matching in PostgreSQL",
  "dcterms:created": "2015-10-28T13:55:15Z",
  "dcterms:modified": "2015-10-28T13:55:15Z",
  "meta:author": "Joe Conway joe.conway@crunchydata.com mail@joeconway.com ",
  "meta:creation-date": "2015-10-28T13:55:15Z",
  "meta:keyword": "",
  "meta:save-date": "2015-10-28T13:55:15Z",
  "modified": "2015-10-28T13:55:15Z",
  "pdf:PDFVersion": "1.5",
  "pdf:charsPerPage": [
    "195",
    "203",
    "144",
    "263",
    "284",
    "276",
    "538",
    "473",
    "454",
{
  "Application-Name": "Microsoft Office Word",
  "Application-Version": "15.0000",
  "Character Count": "19253",
  "Character-Count-With-Spaces": "22585",
  "Content-Length": "302632",
  "Content-Type": "application/vnd.openxmlformats-
      officedocument.wordprocessingml.document",
  "Creation-Date": "2020-08-27T10:15:00Z",
  "Last-Modified": "2020-08-27T10:15:00Z",
  "Last-Save-Date": "2020-08-27T10:15:00Z",
  "Line-Count": "160",
  "Page-Count": "23",
  "Paragraph-Count": "45",
  "Revision-Number": "1",
  "Template": "Thesis_2012.dotx",
  "Word-Count": "3377",
  "X-Parsed-By": [
    "org.apache.tika.parser.DefaultParser",
    "org.apache.tika.parser.microsoft.ooxml.OOXMLParser"
  ],
  "cp:revision": "1",
  "date": "2020-08-27T10:15:00Z",
  "dcterms:created": "2020-08-27T10:15:00Z",
  "dcterms:modified": "2020-08-27T10:15:00Z",
  "extended-properties:AppVersion": "15.0000",
  "extended-properties:Application": "Microsoft Office Word",
  "extended-properties:DocSecurityString": "None",
  "extended-properties:Template": "Thesis_2012.dotx",
  "meta:character-count": "19253",
  "meta:character-count-with-spaces": "22585",
  "meta:creation-date": "2020-08-27T10:15:00Z",
  "meta:line-count": "160",
  "meta:page-count": "23",
  "meta:paragraph-count": "45",
  "meta:save-date": "2020-08-27T10:15:00Z",
  "meta:word-count": "3377",
  "modified": "2020-08-27T10:15:00Z",
  "resourceName": "Thesis Template 2020 v01.docx",
  "xmpTPg:NPages": "23"
}
```

Listing 5: Sample DOCX File Metadata.

7 Database Analysis

This appendix provides detailed information on database analysis done as part of this work. The first section covers the preliminary setup done on the database and following sections are dedicated on covering various query samples. Final section provides suggested improvements to identified issues.

7.1 Database Setup

The database software used in the analysis was PostgreSQL[35] version 13.3. The software was compiled with options given in Listing 6. Once it was installed, it was started and a default database was created to be used in the analysis process.

```
$ wget -q https://ftp.postgresql.org/pub/source/v13.3/postgresql-
    13.3.tar.gz
$ tar xzf postgresql-13.3.tar.gz
$ cd postgresql-13.3
$ CFLAGS=-I/usr/local/include \
     LDFLAGS=-L/usr/local/lib \
     ./configure \
     --prefix=$HOME/testdb/pg13 \
     --with-uuid=bsd
$ gmake && gmake install
$ (cd contrib/dict_int && gmake install)
$ (cd contrib/fuzzystrmatch && gmake install)
$ (cd contrib/pg_trgm && gmake install)
$ (cd contrib/uuid-oosp && gmake install)
$ cd ..
$ ./pg13/bin/initdb -D testdb
$ ./pg13/bin/pg_ctl -D testdb -l logfile start
$ ./pg13/bin/createdb testdb
$ ./pg13/bin/psql testdb
```

Listing 6: Compilation of the Database Software.

For the system data a database dump was taken from the new digital asset management systems staging environment dated 8th of June 2021:

[35] https://www.postgresql.org

```
-rw-r-----  1 tmy  tmy   26419058 Jul 11 13:06 core-staging-20210708.sql
```

The Voikko[36] extension setup required applying a small patch to make it compile as detailed in Listing 7.

```
$ cd postgresql-13.3/contrib
$ git clone https://github.com/Houston-Inc/dict_voikko.git
$ wget https://github.com/zmyrgel/dict_voikko/commit/ea7760.diff
$ patch < ea7760.diff
$ gmake install
```

Listing 7: Compilation of the dict_voikko Extension.

Once the dict_voikko extension was installed the database required few commands for adding Voikko support given in Listing 8.

```
CREATE EXTENSION IF NOT EXISTS dict_voikko;

CREATE TEXT SEARCH DICTIONARY voikko_stopwords (
    TEMPLATE = voikko_template, StopWords = finnish
);

CREATE TEXT SEARCH CONFIGURATION voikko (COPY = finnish);

ALTER TEXT SEARCH CONFIGURATION voikko
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
      word, hword, hword_part
    WITH voikko_stopwords, finnish_stem;
```

Listing 8: Setup Steps for Enabling Voikko Contrib.

7.2 Database Benchmarks

The focus on benchmarking is not to get absolute maximum performance, only to cover rough estimates on how scalable the various search methods are.

For benchmarking how well the database FTS search can keep up with increasing data the following code was used.

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

```

[36] https://voikko.puimula.org

```sql
CREATE TABLE sample_products_1000 (
  id SERIAL PRIMARY KEY,
  name TEXT,
  description TEXT,
  product_code TEXT,
  metadata JSONB,
  tsv_data TSVECTOR,
  UNIQUE(product_code)
);

CREATE TABLE sample_products_10_000 (
  id SERIAL PRIMARY KEY,
  name TEXT,
  description TEXT,
  product_code TEXT,
  metadata JSONB,
  tsv_data TSVECTOR,
  UNIQUE(product_code)
);

CREATE TABLE sample_products_100_000 (
  id SERIAL PRIMARY KEY,
  name TEXT,
  description TEXT,
  product_code TEXT,
  metadata JSONB,
  tsv_data TSVECTOR,
  UNIQUE(product_code)
);

CREATE TABLE sample_products_1_000_000 (
  id SERIAL PRIMARY KEY,
  name TEXT,
  description TEXT,
  product_code TEXT,
  metadata JSONB,
  tsv_data TSVECTOR,
  UNIQUE(product_code)
);

-- Use temporary table to contain system words
CREATE TABLE words (word TEXT);
CREATE TABLE names (name TEXT);

-- Copy data from system dictionary
COPY words (word) FROM '/usr/share/dict/words';
COPY names (name) FROM '/usr/share/dict/propernames';
```

```sql

-- Fill tables with random data
INSERT INTO sample_products_1000 (name, description, product_code)
SELECT
  (SELECT * FROM words TABLESAMPLE SYSTEM (0.5)
   ORDER BY RANDOM()+_g*0 LIMIT 1),
  ARRAY_TO_STRING(ARRAY(SELECT * FROM words
                        TABLESAMPLE SYSTEM (0.5)
                        ORDER BY RANDOM()+_g*0
                        LIMIT CEIL(RANDOM()*10+_g*0)), ' '),
  UUID_GENERATE_V4()
FROM GENERATE_SERIES(1, 1000) AS _g;

INSERT INTO sample_products_10_000 (name, description, product_code)
SELECT
  (SELECT * FROM words TABLESAMPLE SYSTEM (0.5)
   ORDER BY RANDOM()+_g*0 LIMIT 1),
  ARRAY_TO_STRING(ARRAY(SELECT * FROM words
                        TABLESAMPLE SYSTEM (0.5)
                        ORDER BY RANDOM()+_g*0
                        LIMIT CEIL(RANDOM()*10)), ' '),
  UUID_GENERATE_V4()
FROM GENERATE_SERIES(1, 10000) AS _g;

INSERT INTO sample_products_100_000 (name, description,
    product_code)
SELECT
  (SELECT * FROM words TABLESAMPLE SYSTEM (0.5)
   ORDER BY RANDOM()+_g*0 LIMIT 1),
  ARRAY_TO_STRING(ARRAY(SELECT * FROM words
                        TABLESAMPLE SYSTEM (0.5)
                        ORDER BY RANDOM()+_g*0
                        LIMIT CEIL(RANDOM()*10)), ' '),
  UUID_GENERATE_V4()
FROM GENERATE_SERIES(1, 100000) AS _g;

INSERT INTO sample_products_1_000_000 (name, description,
    product_code)
SELECT
  (SELECT * FROM words TABLESAMPLE SYSTEM (0.5)
   ORDER BY RANDOM()+_g*0 LIMIT 1),
  ARRAY_TO_STRING(ARRAY(SELECT * FROM words
                        TABLESAMPLE SYSTEM (0.5)
                        ORDER BY RANDOM()+_g*0
                        LIMIT CEIL(RANDOM()*10)), ' '),
  UUID_GENERATE_V4()
FROM GENERATE_SERIES(1, 1000000) AS _g;
```

```sql

-- fill in the tsvector data
UPDATE sample_products_1000
SET tsv_data =
 setweight(to_tsvector('simple', COALESCE(name, '')), 'A') ||
 setweight(to_tsvector('simple', COALESCE(product_code, '')), 'A')
    ||
 setweight(to_tsvector('simple', COALESCE(description, '')), 'B') ||
 setweight(to_tsvector('simple', COALESCE(metadata, '{}')), 'C');

UPDATE sample_products_10_000
SET tsv_data =
 setweight(to_tsvector('simple', COALESCE(name, '')), 'A') ||
 setweight(to_tsvector('simple', COALESCE(product_code, '')), 'A')
    ||
 setweight(to_tsvector('simple', COALESCE(description, '')), 'B') ||
 setweight(to_tsvector('simple', COALESCE(metadata, '{}')), 'C');

UPDATE sample_products_100_000
SET tsv_data =
 setweight(to_tsvector('simple', COALESCE(name, '')), 'A') ||
 setweight(to_tsvector('simple', COALESCE(product_code, '')), 'A')
    ||
 setweight(to_tsvector('simple', COALESCE(description, '')), 'B') ||
 setweight(to_tsvector('simple', COALESCE(metadata, '{}')), 'C');

UPDATE sample_products_1_000_000
SET tsv_data =
 setweight(to_tsvector('simple', COALESCE(name, '')), 'A') ||
 setweight(to_tsvector('simple', COALESCE(product_code, '')), 'A')
    ||
 setweight(to_tsvector('simple', COALESCE(description, '')), 'B') ||
 setweight(to_tsvector('simple', COALESCE(metadata, '{}')), 'C');

-- index tsvector contents
CREATE INDEX tsv_1k_idx ON sample_products_1000 USING GIN
    (tsv_data);
CREATE INDEX tsv_10k_idx ON sample_products_10_000 USING GIN
    (tsv_data);
CREATE INDEX tsv_100k_idx ON sample_products_100_000 USING GIN
    (tsv_data);
CREATE INDEX tsv_1m_idx ON sample_products_1_000_000 USING GIN
    (tsv_data);

-- update statistics
VACUUM ANALYZE;
```

Listing 9: The Benchmark Preliminary Steps

The index sizes given in Table 19 in the sample benchmark database are reasonable and not too large, so the system appears to be able to scale well for near-future growth.

Table 19: Comparison of Benchmark Table Index Sizes.

Name	Table	Size
tsv_1k_idx	sample_products_1000	512 kB
tsv_10k_idx	sample_products_10_000	3728 kB
tsv_100k_idx	sample_products_100_000	26 MB
tsv_1m_idx	sample_products_1_000_000	246 MB

As the Listing 10 displays, the search for data using FTS is fast enough even when using it with table of one million entries in it.

```
core=#  select count(*) from sample_products_1_000_000 where
    tsv_data @@ 'porismatic:*&!syne:*&nontrier';
 count
-------
    44
(1 row)

Time: 113.514 ms
core=#
```

Listing 10: Full-Text Search on One Million Row Data.

For benchmarking if there would be any accute limits on the FTS use the PostgreSQL user manual contents were indexed into the database.

```
$ java -jar ~/Downloads/tika-app-1.26.jar -t postgresql-12-A4.pdf >
    postgresql-12-A4.txt
$ ls -l postgresql-12*
-rw-r--r--  1 tmy  tmy  13025172 Jan  8 09:27 postgresql-12-A4.pdf
-rw-r--r--  1 tmy  tmy   6455797 Jan  8 09:28 postgresql-12-A4.txt
```

Listing 11: Extracting PostgreSQL Manual Content.

After the manual content was extracted into a text file they can be stored in the database:

```
core=# create temp table tsv (id int, content text, tsv tsvector);
```

```
CREATE TABLE
\set content `cat postgresql-12-A4.txt`
core=# INSERT INTO tsv (id, content) VALUES(1, :'content');
INSERT 0 1
core=# update tsv set tsv = to_tsvector('english', content);
UPDATE 1
core=# CREATE INDEX idx_fts_tsv ON tsv USING gin(tsv);
CREATE INDEX
```

Listing 12: Testing Indexing of the PostgreSQL Manual Contents.

7.3 Database Issues Analysis

There were some identified issues in the use of database which are detailed in this section. Current flawed search term handling done by the DAM application is shown in Listing 13.

```
keywords
  .toLowerCase()            // lowercase string
  .trim()                   // remove leading/trailing whitespace
  .match(/(\b([a-zåäö0-9]+\b))+/g)? // group alphanum sequences
  .join('<->')              // join groups with FOLLOWED BY operator
  .replace(/\s{2,}/g, ' ') // replace consecutive spaces with one
  .split(' ')               // split on space
  .join(':*&') + ':*';      // exclusive join with prefix matching
```

Listing 13: Current Search Query.

The implementation is flawed as it effectively makes the application search to do phrase searches by default. Another problem in the code is that it makes use of prefix matching only for the last term, which does not seem what the author has intended.

There exists many fields in the product metadata, these vary based on product type but full list is given in Listing 14.

```
select jsonb_object_keys(metadata) from products;
------------------------
 containsConnectionInfo
 creator
 dateIssued
 imprint
 owner
 ownerUnit
 subject
 titleAlternatives
 uriDescription
(9 rows)
```

Listing 14: Sample Metadata Keys in Customer Data.

To improve the matching only the “creator”, “imprint”, “subject”, “titleAlternatives”, “uriDescription” fields should be added to the tsvector as these contain textual data. For example “subject” metadata contains product specific keywords which could then be weighted more heavily than rest of the metadata by indexing metadata field separately. As metadata fields are customer-specific this process needs to be done individually for each customer.

The original database trigger run after each product table change was:

7.4 Database Query Analysis

This section provides detailed analysis on running searches on the PostgreSQL database focusing on performance and features. For performance analysis the samples use the simple timing command to give approximate amount of time on how long it takes to complete the query. The commonly used analysis command *EXPLAIN ANALYZE* is not used as its output is very verbose and the additional details are not required for these simple comparisons. The command was used to verify that the queries used correct search plans and utilized table indices.

The product table indices used during database analysis are given in Listing 16.

```
Indexes:
    "products_pkey" PRIMARY KEY, btree (id)
    "products_product_code_key" UNIQUE CONSTRAINT, btree
    (product_code)
    "products_tsv_data_idx" gist (tsv_data)
    "trgm_full_idx" gin (immutable_concat_ws(' '::text, VARIADIC
    ARRAY[product_code::text, name::text, description, metadata ->>
    'subject'::text, metadata ->> 'creator'::text, metadata ->>
    'titleAlternatives'::text]) gin_trgm_ops)
```

Listing 16: Product Table Indexes.

```
CREATE OR REPLACE FUNCTION create_or_update_product_tsv_data()
RETURNS trigger
AS $create_or_update_product_tsv_data$
BEGIN
 UPDATE products
 SET tsv_data =
   setweight(to_tsvector('simple',
     COALESCE(t.name, '')), 'A') ||
   setweight(to_tsvector('simple',
     COALESCE(t.product_code, '')), 'A') ||
   setweight(to_tsvector('simple',
     COALESCE(t.description, '')), 'B') ||
   setweight(to_tsvector('simple',
     COALESCE(t.metadata, '{}')), 'C') ||
   setweight(to_tsvector('simple',
     COALESCE(regexp_replace(t.o_username, \\
      '\\@|\\+|\\.', ' ', 'gim'), '')), 'A') ||
   setweight(to_tsvector('simple',
    COALESCE(t.o_first_name, '')), 'A') ||
   setweight(to_tsvector('simple',
    COALESCE(t.o_last_name, '')), 'A')
 FROM ( SELECT p.*,
               owner.username AS o_username,
               owner.first_name AS o_first_name,
               owner.last_name AS o_last_name
        FROM products p
        LEFT JOIN public.users owner ON owner.id = p.created_by
        WHERE CASE TG_TABLE_NAME
              WHEN 'products'
              THEN p.id = NEW.id
              END
      ) t
 WHERE products.id = t.id;
 RETURN NEW;
END;
$create_or_update_product_tsv_data$ LANGUAGE plpgsql;
```

Listing 15: Original Product TSVector Update Trigger.

In order to index string data for trigram matching the database requires an immutable version of string concatenation function. Immutable in here means that the output of the function will not vary based on the user specified locale settings as is the case with the default concatenation function. The default concatenation function works with dates as well which output format is heavily dependent on the locale.

```
CREATE OR REPLACE FUNCTION immutable_concat_ws(text, VARIADIC
    text[])
RETURNS text AS 'text_concat_ws' LANGUAGE internal IMMUTABLE
    PARALLEL SAFE;
```

Listing 17: Sample Concat Utility.

The first query is matching pattern “kissa” from within the searchable fields in Listing 18.

```
SELECT p.name, p.product_code, p.description, p.metadata,
       u.username, u.first_name, u.last_name
FROM products p
INNER JOIN users u ON p.created_by = u.id
WHERE concat_ws(p.name, p.product_code, p.description, p.metadata,
       u.username, u.first_name, u.last_name) ILIKE '%kissa%';
```

Listing 18: Database Query with Pattern Matching.

A sample of using trigram matching to match documents with term “kissa” in Listing 19. The query starts to become long as the trigram query requires concatenating the fields together to big string value to be able to use indexes.

```
SELECT p.name, p.product_code, p.description, p.metadata,
       u.username, u.first_name, u.last_name,
       word_similarity('kissa', concat_ws(
         p.name, p.product_code,
         p.description, p.metadata,
         u.username, u.first_name, u.last_name)
         ) as score
FROM products p
INNER JOIN users u ON p.created_by = u.id
WHERE 'kissa' <% concat_ws(p.name, p.product_code, p.description,
       p.metadata, u.username, u.first_name, u.last_name)
ORDER BY word_similarity('kissa',
  concat_ws(p.name,p.product_code,p.description,p.metadata,
            u.username, u.first_name, u.last_name)) DESC;
```

Listing 19: Database Query with pg_trgm Extension.

The simple FTS search is presented in Listing 20. The query is just matching the term “kissa” from the tsvector field containing the pre-calculated document terms.

```
core=# SELECT p.id, p.name, p.product_code, p.description,
       p.metadata, u.username, u.first_name, u.last_name
FROM products p
INNER JOIN users u ON p.created_by = u.id
```

```
WHERE p.tsv_data @@ 'kissa'
22 rows
```

Listing 20: Database query With Full-Text Search.

The improved FTS in Listing 21 is using prefix-matching to get better results.

```
core=# SELECT p.id, p.name, p.product_code, p.description,
       p.metadata, u.username, u.first_name, u.last_name
FROM products p
INNER JOIN users u ON p.created_by = u.id
WHERE p.tsv_data @@ 'kissa:*'
(39 rows)
```

Listing 21: Database Query with Full-Text Search with Prefix Matching.

Example of how pg_trgm may be used with multiple terms such as "kissa AND koira" is given in Listing 22. The query is using weighted scoring to improve ranking of the results which results in quite complex query.

```
 SELECT p.id, p.name,
        (word_similarity('kissa', p.product_code) +
         word_similarity('kissa', p.name) +
         word_similarity('kissa', p.description) * 0.75 +
         word_similarity('kissa',
           COALESCE(p.metadata->>'subject', '0')::text) +
         word_similarity('kissa',
           COALESCE(p.metadata->>'creator', '0')::text) +
         word_similarity('kissa',
           COALESCE(p.metadata->>'titleAlternatives', '0')::text) *
    0.75)
         +
        (word_similarity('koira', p.product_code) +
         word_similarity('koira', p.name) +
         word_similarity('koira', p.description) * 0.75 +
         word_similarity('koira',
           COALESCE(p.metadata->>'subject', '0')::text) +
         word_similarity('koira',
           COALESCE(p.metadata->>'creator', '0')::text) +
         word_similarity('koira',
           COALESCE(p.metadata->>'titleAlternatives', '0')::text) *
    0.75 ) as score
FROM products p
INNER JOIN users u ON p.created_by = u.id
WHERE ( 'koira' <% p.product_code
     OR 'koira' <% p.name
     OR 'koira' <% p.description
     OR 'koira' <% (p.metadata->>'subject')::text
```

```
        OR 'koira' <% (p.metadata->>'creator')::text
        OR 'koira' <% (p.metadata->>'titleAlternatives')::text)
    AND ( 'koira' <% p.product_code
        OR 'koira' <% p.name
        OR 'koira' <% p.description
        OR 'koira' <% (p.metadata->>'subject')::text
        OR 'koira' <% (p.metadata->>'creator')::text
        OR 'koira' <% (p.metadata->>'titleAlternatives')::text)
        ORDER BY score DESC;
```

Listing 22: Sample Multiterm Query Using Trigram Matching with Ranking.

Compared the trigram version in Listing 22 the FTS version shown in Listing23 is simple.

```
SELECT p.name, p.product_code, p.description, p.metadata,
       u.username, u.first_name, u.last_name,
       ts_rank(tsv_data, 'kissa:* & koira:*') as rank
FROM products p
INNER JOIN users u ON p.created_by = u.id
WHERE tsv_data @@ 'kissa:* & koira:*'
ORDER BY rank DESC;
```

Listing 23: Sample Multiterm Query Using Full-Text Search With Ranking.

The searching file contents with trigrams will be very slow as presented in Listing 24. The primary cause is in trigrams, the search needs to iterate through each term and calculate their trigrams and then compare them with search term. These kinds of searches should be reserved for smaller text sizes.

```
core=# create index trgm_content_idx on renditions USING GIN
    (file_content gin_trgm_ops);
CREATE INDEX
Time: 243279.278 ms (04:03.279)
core=#
core=# select count(id) from renditions where 'kissa' <%
    file_content;
 count
-------
  4248
(1 row)

Time: 298336.125 ms (04:58.336)
```

Listing 24: Trigram Searching File Content.

Searching file contents using regular pattern matching query in Listing 25 shows that it is slightly faster in execution than trgm search in Listing 24 but it is too slow to fit within the constraints.

```
core=# select count(id) from renditions where file_content ilike
    '%kissa%';
 count
-------
  1839
(1 row)

Time: 14859.720 ms (00:14.860)
```

Listing 25: Pattern Matching for File Content.

The use of FTS is shown in Listing 26. It starts by extending the table with tsv_contents field which is then filled with weighted tsvector contents. This allows each row to store the calculated tsvector contents, so it does not need to be calculated for each query. The first select query is run without using index and is too slow to fit within the time constraints. Once the index is added the query executes well within the time constraints making it the only search option to do so.

7.5 Database Improvements

This section provides details on the suggested improvements for the DAM system.

A suggested improvement for the original database trigger presented in Listing 15 is given in Listing 27. It improves the original by using language-specific dictionary configurations and indexing the username field as-is.

```
 CREATE OR REPLACE FUNCTION create_or_update_product_tsv_data()
    RETURNS trigger
 AS $create_or_update_product_tsv_data$
 BEGIN
   UPDATE products SET tsv_data =
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.name, '')), 'A') ||
     setweight(to_tsvector('simple',
       COALESCE(t.product_code, '')), 'A') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.description, '')), 'B') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.metadata->>'subject', '')), 'A') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.metadata->>'creator', '')), 'A') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.metadata->>'titleAlternatives', '')), 'B') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.metadata->>'imprint', '')), 'C') ||
     setweight(to_tsvector(t.trg_dict::regconfig,
       COALESCE(t.metadata->>'uriDescription', '')), 'C') ||
     setweight(to_tsvector('simple',
       COALESCE(t.o_username, '')), 'A') ||
     setweight(to_tsvector('simple',
       COALESCE(t.o_first_name, '')), 'A') ||
     setweight(to_tsvector('simple',
       COALESCE(t.o_last_name, '')), 'A')
   FROM (
     SELECT p.*,
               owner.username AS o_username,
               owner.first_name AS o_first_name,
               owner.last_name AS o_last_name,
               CASE WHEN new.content_lang = 1 THEN 'finnish'
                      WHEN new.content_lang = 2 THEN 'swedish'
                      WHEN new.content_lang = 4 THEN 'english'
                      ELSE 'simple'
               END trg_dict
      FROM products p
      LEFT JOIN public.users owner ON owner.id = p.created_by
      WHERE CASE TG_TABLE_NAME
               WHEN 'products' THEN p.id = NEW.id
               END
  ) t
  WHERE products.id = t.id;
  RETURN NEW;
END;
$create_or_update_product_tsv_data$ LANGUAGE plpgsql;

CREATE TRIGGER create_or_update_tsv_data
AFTER INSERT OR UPDATE OF name, description, product_code, metadata
    ON products
FOR EACH ROW
EXECUTE PROCEDURE create_or_update_product_tsv_data();
```

Listing 27: Suggested Improvement for the TSVector Update Trigger.

For looking at the problem of parsing hyphenated strings with the FTS parser in PostgreSQL returns a bit of varied results regarding numbers in hyphened strings:

```
core=# select to_tsvector('simple', 'abc-100');
    to_tsvector
-------------------
 '-100':2 'abc':1
 (1 row)
core=# select to_tsvector('simple', 'a-abc-100');
           to_tsvector
---------------------------------
 '100':4 'a':2 'a-abc':1 'abc':3
```

Listing 28: Fixing Product Code Hyphenation Parsing.

The fix is simple with later PostgreSQL versions:

```
CREATE TEXT SEARCH DICTIONARY core_product_code_dict (TEMPLATE
intdict_template, MAXLEN = 64, REJECTLONG = true, ABSVAL = true);
CREATE TEXT SEARCH CONFIGURATION core_product_code (COPY =
pg_catalog.simple);
ALTER TEXT SEARCH CONFIGURATION core_product_code
ALTER MAPPING FOR int, uint WITH core_product_code_dict;

core=# select * from to_tsvector('simple', 'abc-100');
   to_tsvector
------------------
 '-100':2 'abc':1
(1 row)

core=# select * from to_tsvector('core_product_code', 'abc-100');
   to_tsvector
-----------------
 '100':2 'abc':1
(1 row)
```

Listing 29: Fixing Product Code Hyphenation Parsing.

In the first case parsing “abc-100” the parser assumes the number is negative one while in the latter where the string has a “a-” prefix the number is parsed as positive. One of the main search terms used in the DAM system is searching by product code and those use short hyphen separated characters and numbers in them by default. When user would search the above code “abc-1000” with just number “1000” it would not yield match. This particular issue can be fixed by upgrading the PostgreSQL database version to version 13.0 or newer as those include “absval” option for dict_int to force tokenization to return absolute values. This could be used to create separate parser for product codes which would fix this issue.

For sampling how stemming works in practice a sample string of “Koira- ja kissavakuutushakemus” is used as it highlights how various stemming methods work and these are given in Table 20.

Table 20: Comparison of Stemming Algorithm Results.

Dictionary	Tokens
simple	’ja’:2 ’kissavakuutushakemus’:3 ’koira’:1
finnish	’kissavakuutushakemus’:3 ’koira’:1
voikko	’hakemus’:3 ’kissa’:3 ’koira’:1 ’vakuuttaa’:3 ’vakuutus’:3

The FTS use reduces the amount of data needed to be stored in the database as shown below. The main causes are the removal of stop words which remove many filler words from the indexed texts. Other is the removal of duplicates, the FTS only needs to know the place of each indexed term so it only needs to store one copy of the word. Simple comparison of various stemming algorithms on how they affect the word counts is given in Table 30.

Use of any stemming reduces the word count from the original text. The voikko dictionary increases the word count a bit compared to other dictionaries, but this is expected as voikko separates compound words in to individual search terms.

7.6 Database Advanced Query Analysis

This appendix provides details how the PostgreSQL database may be used to implement the more advanced queries familiar in search engines.

7.6.1 Facets

An example of using faceted search by leveraging JSON and window functions modeled after the blog post by Alexander Korotkov [37].

```
WITH all_products AS (
    SELECT
        product_code,
        name,
```

[37] https://akorotkov.github.io/blog/2016/06/17/faceted-search/

```
        product_type_id,
        RANK() OVER (
            PARTITION BY product_type_id
            ORDER BY ts_rank_cd(tsv_data,
    plainto_tsquery('kissa')), id
        ) rank,
        COUNT(*) OVER (PARTITION BY product_type_id) cnt
    FROM products
    WHERE tsv_data @@ plainto_tsquery('kissa')
),
lst AS (
    SELECT
        product_type_id,
        jsonb_build_object(
            'count', cnt,
            'results', jsonb_agg(
                jsonb_build_object(
                    'product_code', product_code,
                    'name', name
        ))) AS data
    FROM all_products
    WHERE rank <= 5
    GROUP by product_type_id, cnt
)
SELECT  jsonb_object_agg(product_type_id, data) FROM lst;
```

Listing 31: Example Facet Implementation for PostgreSQL.

The above returns by product type how many matching results were found and top 5 results for each group.

7.6.2 Autocomplete

A custom tokenizer for generating edge n-grams was detailed in StackOverflow[38] which can be used to generate n-grams for ts_vector:

Once the all the n-grams are stored they can be queried with FTS. One possible improvement would be to limit the n-gram generation to first 20 characters of each word as was done in Bynder as shown in Appendix 5.

[38] https://stackoverflow.com/questions/56894979/edge-ngram-search-in-postgresql

```
core=# ALTER TABLE renditions ADD COLUMN tsv_contents tsvector;
ALTER TABLE
core=# UPDATE renditions r SET tsv_contents =
     setweight(to_tsvector('simple', COALESCE(r.ext_filename, '')),
    'A') ||
     setweight(to_tsvector('simple', COALESCE(r.version_id, '')),
    'A') ||
     setweight(to_tsvector('simple', COALESCE(r.metadata->>'notes',
    '')), 'A') ||
     setweight(to_tsvector('simple', COALESCE(r.file_content, '')),
    'C') ||
     setweight(to_tsvector('simple', COALESCE(r.file_metadata::text,
    '')), 'C');
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
... < previous two lines repeated x 16 > ...
NOTICE:  word is too long to be indexed
DETAIL:  Words longer than 2047 characters are ignored.
UPDATE 18120
Time: 925968.930 ms (15:25.969)
core=#
core=# select count(id) from renditions where tsv_contents @@
    'kissa';
 count
-------
   347
(1 row)

Time: 3150.688 ms (00:03.151)
core=# CREATE INDEX tsv_contents_idx ON renditions USING GIN
    (tsv_contents);
CREATE INDEX
Time: 68122.068 ms (01:08.122)
core=# select count(id) from renditions where tsv_contents @@
    'kissa';
 count
-------
   347
(1 row)

Time: 5.903 ms
core=#
```

Listing 26: Full-text Search on File Contents.

```
core=# select id,
    array_length(regexp_split_to_array(trim(file_content), E'\\W+'),
    1) as words,
       length(to_tsvector('simple', file_content)) as simple,
       length(to_tsvector('finnish', file_content)) as finnish,
       length(to_tsvector('voikko', file_content)) as voikko
from renditions
where id <> 7 order by id
limit 10;
 id | words | simple | finnish | voikko
----+-------+--------+---------+--------
  1 |   107 |     89 |      81 |    103
  2 |   270 |    208 |     175 |    182
  3 |  1351 |    722 |     660 |    625
  4 |  2692 |   1162 |     934 |    817
  5 |  2830 |   1222 |     972 |    836
  6 |  2749 |   1174 |     935 |    812
  8 |  1542 |    715 |     687 |    686
  9 |  3424 |   1106 |    1038 |   1040
 10 |  3322 |   1070 |    1006 |   1003
 11 |     7 |      5 |       5 |      5
(10 rows)
```

Listing 30: Comparison of Word Counts With Stemming.

```
CREATE OR REPLACE FUNCTION edge_gram_tsvector(text text) RETURNS
    tsvector AS
$BODY$
BEGIN
    RETURN (select array_to_tsvector(
             (select array_agg(distinct substring(lexeme for len))
    from unnest(to_tsvector(text)),
             generate_series(1,length(lexeme)) len)
           ));
END;
$BODY$
IMMUTABLE
language plpgsql;
```

Listing 32: Sample Edge N-gram Generator for PostgreSQL.