Spring 2015

INDEX STRATEGIES FOR EFFICIENT AND EFFECTIVE ENTITY SEARCH

Huy T. Vu
San Jose State University

INDEX STRATEGIES FOR EFFICIENT AND EFFECTIVE ENTITY SEARCH

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirement of the Degree

Master of Science

by

Huy T. Vu

May 2015

The Designated Thesis Committee Approves the Thesis Titled

INDEX STRATEGIES FOR EFFICIENT AND EFFECTIVE ENTITY SEARCH

by

Huy T. Vu

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2015

Dr. Thanh Tran Department of Computer Science

Dr. Tsau-Young Lin Department of Computer Science

Dr. Chris Tseng Department of Computer Science

ABSTRACT

The volume of structured data has rapidly grown in recent years, when data-entity emerged as an abstraction that captures almost every data pieces. As a result, searching for a desired piece of information on the web could be a challenge in term of time and relevancy because the number of matching entities could be very large for a given query. This project concerns with the efficiency and effectiveness of such entity queries. The work contains two major parts: implement inverted indexing strategies so that queries can be searched in minimal time, and rank results based on features that are independent of the query itself.

ACKNOWLEDGEMENTS

I would like to thank my family for their great support throughout my study at San Jose State University

I would like to express my gratitude to Dr. Thanh Tran, my thesis advisor, for his supports and motivations in the completion of this project

I would like to thank my project committee member, Dr. Tsau-Young Lin and Dr. Chris Tseng, for their contribution in the completion of this project

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

The outbreak of information technology in the 21st century have caused exponential growth of data in a short time period. Information are collected from very aspects making the volume of raw data extremely large. Social networks alone process about a petabyte of data daily. The level is about the same as governmental and private sector's database. In addition, live streaming data are collected every second. The problem is that despite tremendously huge amount of data, only a very tiny portion of them are used at a certain time [7].

Data scientists have conducted numerous researches on methods to deal with raw collected data, how they should be processed and stored in a way that once they are requested, they are available in a reasonable amount of time. Then, another question arises: given a huge amount of data, and thus high chances that a large number of matching information can be found; in such case, how can data be ranked to ensure that the highest relevant will get higher attention from users.

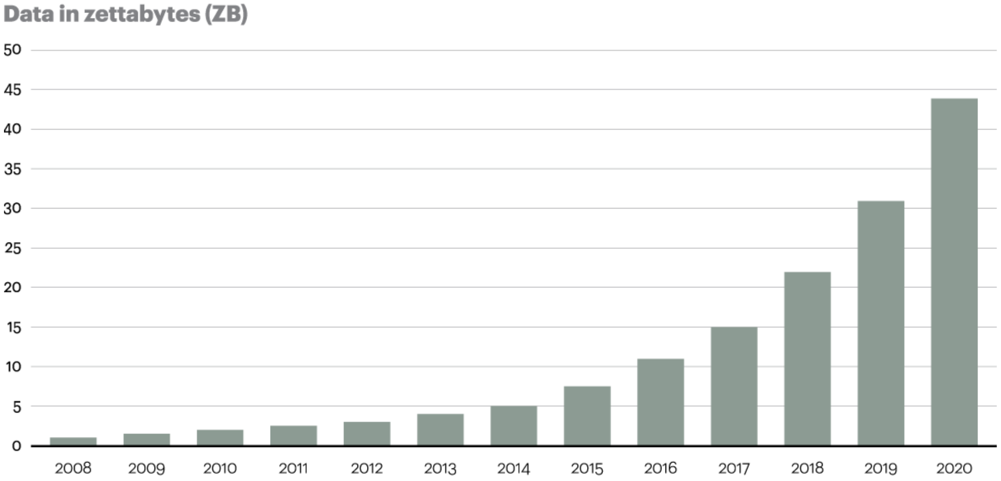


Figure 1: Growth of data, projection into year 2020 [7]

This project offers a solution to such problem for entity search. In the scope of this project, all data are structured entities that represent real-life objects. The main goals are to get result as fast as possible and as accurate as possible given a query.

This project examines three different indexing strategies that were introduced in recent researches on big data mining, an interesting topic that has a lot of developing potentials. This project extends one aspect of these indices and observes how this modification can benefit from the original design.

This project examines three different ranking solutions and introduces an implementation of query-independent features in entity search. Several tactics to store such features are discussed, but the question of which way to store independent features would yield better solution is not within the scope of this paper since it depends on other setups such as engine configurations and local machine performance.

The system is evaluated with practical datasets, large enough to simulate how professional search engine would work in a minimized scale. Several metrics are tested to compare performances of the chosen strategies and scoring schemes.

This project aims to extend functionalities of Apache Solr, a popular search framework that is built on top of Apache Lucene. The goal is to create a plugin to Lucene indexer and Solr searcher that can index and search entities using their designated methods.

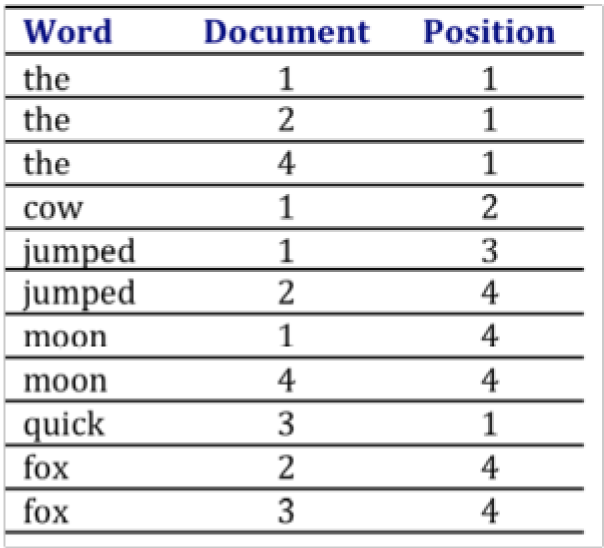
Other concerns regard entity search is not within the scope of this project and therefore will not be discussed in the paper.

2. RELATED WORKS

2.1. Inverted Index

Database Index is a type of data structure that is implemented into the database to enhance the retrieving process. The main purpose of indexing is to quickly locate data without using brutal force to search every entry. In a typical relational database, index is usually created based on columns

Inverted index is a method of database indexing that maps the content to the object it belongs to. This strategy requires extra processing during insertion to database, but allow fast full text searching. In system that data are heavily filled with text, this type of index is shown to be efficient. Generally, inverted index has two variants: record level and word level. This project focus on the latter, which contains not only the reference to documents for each word, but also their location in respective document [2].



Word	Document	Position
the	1	1
the	2	1
the	4	1
cow	1	2
jumped	1	3
jumped	2	4
moon	1	4
moon	4	4
quick	3	1
fox	2	4
fox	3	4

Figure 2: Illustration of inverted index

An extension of inverted index has a feature call payload, an internal storage associated with each mapping. For instance, when a word in mapped to the document it appears in, additional information can be saved such as the exact position of the word, or the relative importance of the word to the content as a whole. Information in the payload is usually used for ranking enhancement purpose and it does not speed up the retrieval process.

There have been previous researches on web search engine algorithm for RDF data retrieval, one of which is to index parallel text with alignment operator to avoid ambiguous meaning of query terms. The idea aims to make indexing structure can be built using a single MapReduce operation.

This project concerns three designs of inverted index, which are introduced below. For a better illustration, consider this sample entity to see how it would be indexed using different designs

Title: San Jose State University
Region: Northern California
Education: coed
Sector: public
Academic: four-year
Term: semester

Figure 3: A sample entity object represented as document in Lucene

2.1.1. Vertical Indexing

The term “vertical” reflect the by-column nature of this strategy. Indeed, this indexing scheme is a straightforward design, in which an index term field is created for each property of an entity. This requires extra storages for multiple indices, but grants amazing fast query time as the engine simply needs to check the index correspond to the fields that the query is interested in. Another advantage of this strategy is its ability to restrict matching to a specific field. For example, field “genre” of entity “All Rise” (a song) and field “color” of entity “Pacific Ocean” (an ocean) could contain the same value “BLUE”, under this indexing, querying on “Blue” can distinguish the two entities by giving the specific field to search.

Table 1: Illustration of vertical design of inverted index

Index	**Value**
Title	San Jose State U.
Region	Northern California
Education	Coed
Sector	Public
Academic	Four-year
Term	Semester

2.1.2. Horizontal Indexing

This indexing scheme requires much less indices than the previous. Only two indices are required: one to hold names of all fields of entities, and the other to carry values of corresponding fields. Since the number of indices is constant-space complexity, this strategy is somewhat more appealing than the previous. In certain dataset where data also contain URI or structural properties, an additional field could be added to store the anchor text or the extra information [2].

However, horizontal indexing has a downside: it could create ambiguity. This is because each term field of horizontal indexing needs to store multiple values, and thus ambiguity is inevitable. For instance, if there are two fields in a document contain the same keyword, then when that keyword is search, the system may misrecognize it as value of the other field.

Table 2: Illustration of horizontal design of inverted index

Index	**Value**
Fields	Title, Region, Education, Sector, Academic, Term
Tokens	San Jose State University, Northern California, Coed, Public, Four-year, Semester

2.1.3. Reduced Indexing

This schema takes the advantages of the previous two to improve performance. Starting out similar to vertical schema, Reduced indexing, provides more flexibilities by grouping different indices in vertical design based on their level of importance [2]. The scale is defined and adjusted for specific use. In this project, three levels are used, denoting fields that are very important, neutral, and unimportant. Once grouped together, information of fields in each group is stored similarly to the horizontal scheme. An obvious advantage over vertical indexing is faster access during query time. However, a big drawback of this method is the limitation of functionality, as it is not effective for query that restricts matching to a particular fields

Table 3: Illustration of reduced design of inverted index

Index	**Value**
Important	Title: San Jose State University, Academic: Four-year
Neutral	Sector: Public, Region: Northern California
Unimportant	Education: Coed, Term: Semester

This design clearly is the combination of the previous two. In addition, it introduces another degree of independence, called the reduced indexing factor. It notes the ability to decide how many categories to implement and which property falls into which category. Standards for such decision vary depends on the nature of data as well as the purpose of the search engine. In practice, there could be more than three levels of importance if the field space is large, and the number is between 1 and number of fields [9]. Theoretically, the number of levels should assume some proportional relations with the number of fields. This project introduces some ways to select the number of fields for reduced strategy, but an algorithm for determining the most effective division is out of scope of this paper.

2.2. Learning to Rank

Learning to rank is a technique used in many applications for information retrieval, especially in machine-learned search engine. Due to large volume of data, a two-stage ranking is often used to enhance retrieving speed. First, a much smaller portion of potential matching entities are fetched from database using simple models, making a top-N retrieval; then a much more complex ranking algorithm is used to determine the final results [4]. This sections introduces several simple weighing schemes that are used directly or indirectly in the project.

2.2.1. TF-IDF

Term frequency – Inverse document frequency, often referred to as TF-IDF, is a numerical statistic indicates how significance an element is to a collection of elements, often used as a weighing factor in text mining application. TF-IDF is variant among search engines but the following basic principle must be satisfied: its value should always be directly proportional to the number of occurrence of the element in the collection.

Term frequency (TF) is defined as the number of occurrence of an element (in this project, element is defined as a single word, and the collection is text that could be correspond to multiple subject fields by default, or to a specific field) in the collection. For a query phrase, term frequency is simply the sum of appearance frequency of each word. In longer documents, augmented term frequency is used to prevent bias, utilizing double normalization concept. This normalization is often used in today search engine, with some variants to suit specific data types.

$$TF(q_i, D) = 0.5 + 0.5\frac{f_{q_i,D}}{\max(f_D)}$$

Nevertheless, some common words such as articles and transitions appear with overwhelming frequencies, while do not semantically contribute to the query. This often skews the term frequency calculation, as meaningful terms are less emphasized compared to common English words such as “the, therefore, such…” Hence, the concept inverse document frequency (IDF) is introduced to reflect the level of significance of the information provided by the keywords. It tells whether a word is common or rare, and through which it makes adjustments by increasing weights of rarely apparent words while significantly decreasing or diminishing the weights of words that have tremendously high appearance rate.

Generally, the IDF weight of a term is calculated by dividing the total count by number of documents containing that term. However, just as TF calculation can become biased in larger documents, IDF could be affected by large volume of documents and hence, a logarithmic normalized factor is applied.

$$IDF(q_i) = log\frac{N - |d(q_i)| + 0.5}{|d(q_i)| + 0.5}$$

$$where\ N\ is\ total\ docs$$

$$|d|\ is\ number\ of\ docs\ contain\ term\ q$$

The final weight computation for TF-IDF is their product:

$$weight_{TF.IDF} = TF \times IDF$$

2.2.2. Okapi BM25

BM25 is a ranking function that sorts documents according to their relevance to a specific query, first implemented with the Okapi information retrieval system. This function ranks a set of documents based on the appearance of query terms in each document. Since keywords are compared independently, the total weight is simply the score summation [1].

$$score(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{TF(q_i, D) \cdot (k+1)}{TF(q_i, D) + k(1 - b + \frac{b \cdot |D|}{avgDl})}$$

$$with\ Q\ contains\ q_1, q_2, \dots q_n$$

$$D\ denotes\ the\ interested\ document$$

$$and\ avgDl\ is\ the\ averge\ length\ of\ all\ documents$$

BM25 employs two free parameters *k and b* that distinguish it from other ranking methods. Note from the equation that b serves as a normalization probabilistic factor and thus has value from 0 to 1. In practice, k is often within range [1.2, 2.0] and b = 0.75 for optimization.

2.2.3. BM25F and other variants

Blanco and Mika introduces BM25F, a variant of the Okapi ranking function that takes structures and anchor texts into account. This is an important feature because it could be used to establish relations between entities into entity graphs, which is useful as today data on the web are highly linked [4].

Other variants include BM11 (when b=1) and BM15 (when b=0) at the extreme end of b. They are the original designs of the BM weighing scheme, but are not competent with the current BM25

2.3. Query-Independent Features

Besides the traditional ranking based on content relevance, there are other approaches that stands independent from the query. One popular example is the use of centrality, taking advantage of graph-like relationship of data. This method uses algorithms such as PageRank to compute a global score for any page denoting the likelihood of a user entering that page while surfing the nearby pages. In reality, a more simple solution is often preferred. Features that are based on frequency can be used to count the popularity of nodes and edges in the entity graphs [4], [15].

This project employs simple but mostly seen independent features: the recency and popularity of an entity. Since they are independent factors, they could be designed to serve solely the dataset. Most of the time, recency denotes the document age or the time period after a document is indexed and before it is queried. In this project, recency calculation is inspired by a study on query effectiveness optimization and therefore it is not a linear value, but as a reciprocal function of the document age [14].

$$recency = \frac{a}{mx + b}$$

$$where\ x\ is\ the\ document\ age\ \&\ \ a, m, b\ are\ tuning\ parameters$$

Popularity is a sensitive parameter that could change constantly, and is often loosely defined. It can be the click-through rate, or the frequency at which an entity is requested. Nevertheless, this quantity should be normalized to avoid skew result. For instance, a web page that has few visits and a colossal site with millions of subscribers, their native true value would result in skew computation. A common method to prevent this is normalizing them with their maximum value.

3. PROJECT DESIGN

3.1. Definition

3.1.1. Problem Formulation

Given a large volume of data, how to store indices and formulate a ranking method such that when a query is triggered, results are returned in an appropriate order within minimal amount of time

3.1.2. Terminology

The following terms are widely used in the report:

- *Entity*: a concept or abstract that has a complete meaning by itself. In this project, entity represents an object with unique id and properties. Entity may include, but not limited to persons, subjects, records, concepts…
- *Document*: a concept of Lucene and Solr, represents a unit of data object or a single entity. Documents of the same type should have uniform set of fields, even though not all of them may contain a meaningful value
- *Field/Property*: an attribute of an entity. A field can either be indexed, stored or both. It can contain multiple values for one entity, in which case it may be best represented with a term vector.
- *Token/Value*: value of a field/property. It can assume any type, from primitive type such as number, string, date… to more complicated ones. This this project, besides number and date, all other types are modified on Lucene base type to meet the goal.

- *Reduced Factor*: Number of defined levels for reduced indexing. It is default to 3 (important, neutral and unimportant) but with larger pools of fields, this factor can increase to prevent the index from becoming too horizontal. This project limits the range from 3 to 5.
- *Default Field*: a virtual field that exists only to hold all values of the entity. It is used as a search field when a query does not specify any particular field
- *Query*: a user-input phrase in English that is passed into the search engine. This raw query will be processed into a format readable by the search engine. A query can simply be a plaintext that searches over default location or a structured format with specified properties
- *Similarity*: denotes the relevancy between an entity and a query, as a numerical value computed by a similarity functions. The higher the value, the more closely an entity relates to the query. In this project, similarity can be used interchangeably with relevancy
- *Recency*: the concept of entity age, which denoted by the chronological difference from the entity's indexing its retrieval moment. In this project, recency is regarded as a discount factor that put less emphasis on older entities. The highest value of this factor is 1 for recently indexed entities, while the very old have their recency value close to 0.
- *Popularity*: the concept that measures the credential of the entity, how popular is a particular entity compared to the common ground of all other entities. This simulates the fact in reality that certain sources of information are more valuable to others based on just the source identity itself. Popularity is a relative

measurement, ranging from 0 to 1, directly proportional to the usefulness of an entity. Popularity gives independent information

3.2. Technology

3.2.1. Apache Lucene

Lucene is an open source information retrieval library, a powerful full-text search engine for cross-platform applications. Lucene has been widely used to implement Internet search engines, usually for single-site searching. In Lucene, entities are stored as documents characterized by a number of fields. Document indices can be stored in a single location or across multiple shards, in which requires the system to run in cloud mode.

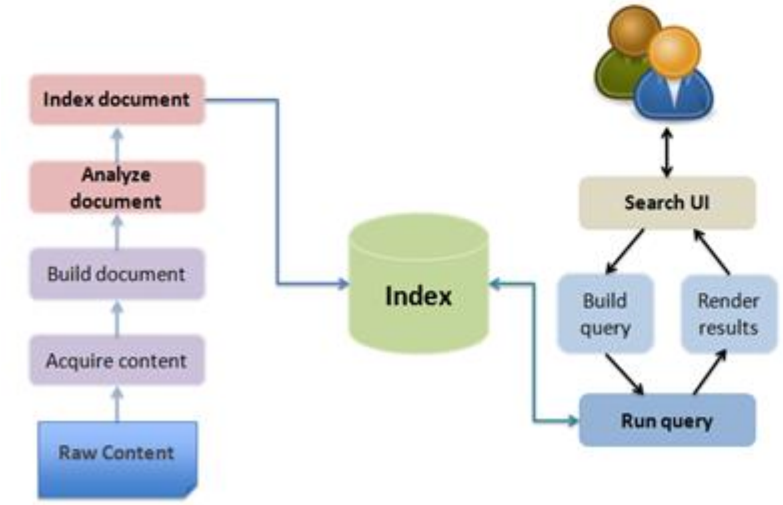


Figure 4: Lucene architecture and flow diagram

3.2.2. Solr-Java API

Solr is a high-performance search platform based on core Lucene, and among the most popular enterprise solution. Among its major features, full-text search and real-time indexing are essential for this project. In addition, Solr also provides distributed search and index replication, which is useful for running in scalable systems [16].

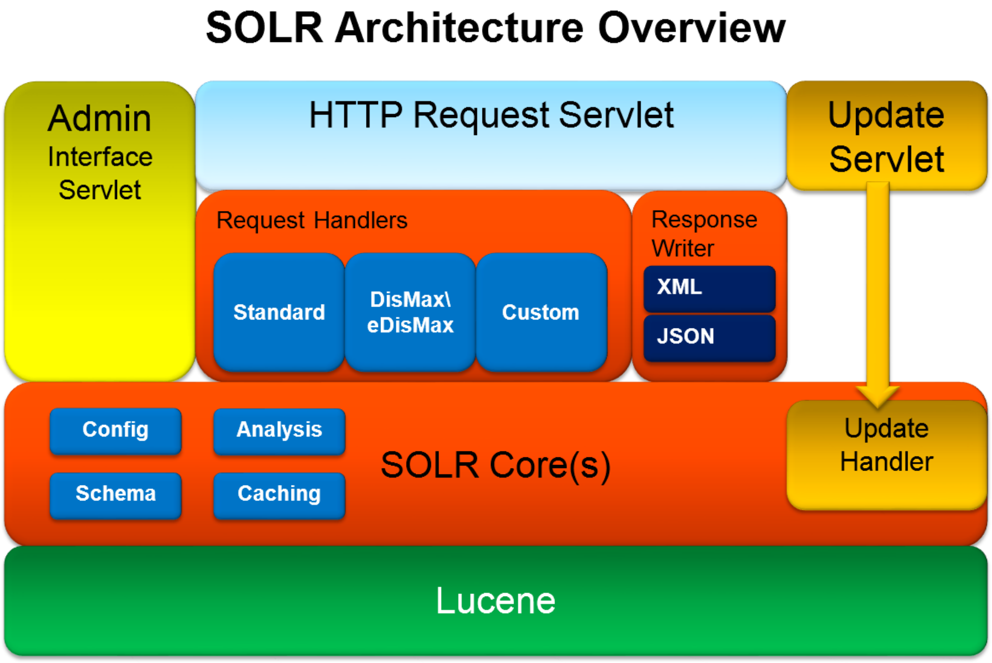


Figure 5: Solr architecture

This project uses SolrJ, an API for Java clients to access Solr. The library allows development

3.3. Query Efficiency

Query efficiency is a performance metric to measure how fast a search engine can retrieve data. Retrieval time is defined as the period after the query is accepted into the search engine until the result is returned.

This project concerns three indexing strategies: vertical, horizontal, and reduced, which definitions and logics have been introduced in the previous section. Retrieval time shall be recorded to compare efficiency of each strategy.

While vertical and horizontal indexing strictly follow their definitions, there are some flexibility in reduced indexing strategy. This project hypothesizes that larger set of fields in a schema should be classified into more categories to maintain balances. Since Reduced index is a mixture of both vertical and horizontal indices to take advantages of both, it should be consistently balanced. Too small and it will acts like horizontal index, while too large will make it behave like vertical. This project measure efficiency limiting the range from 3 to 5.

3.4. Query Effectiveness

Query effectiveness is a performance metric to measure how relevant is the search result to the query terms. This project considers some solutions in which documents are ranked against each other.

3.4.1. Simple Relevancy Score

This is the standard ranking in any search engine. The scheme uses the classic BM25F similarity function to weigh entities based on their relevance with the query terms. This gives high accurate ranking of the results at the cost of computing TF-IDF for each query. Results from this ranking solution are compared with those with query independent features for their performance in term of effectiveness.

3.4.2. Independent Features

This scheme combines the weight computed by BM25F, and two additional features that are independent from the query. Recency and popularity are features that associated with the entities as soon as they are indexed into the database. They are pre-computed and stored along with other features so that when the entity is retrieved, the system does not need to compute values for popularity and recency again. The goal is utilizing pre-computed values of each document to rank them, which could boost up the process by a small fraction of time. In practice, every little time saved by employing this mechanism could accumulate into a huge effectiveness.

There are two ways to employ pre-computed independent features into the document index. The simple one is to store the information using payload that is directly associated with the term. This information can be obtain easily by the scorer. Another way is to use extra fields, specifically a date field to store recency and an external field to store popularity (so that it can be constantly updated). This method proposes a simple way to store information at index time, but at query time, it is difficult to obtain the information from the fields. To overcome this, the field can stored as a pseudo-payload to the actual index, so that the information can be extracted for scoring purpose.

4. IMPLEMENTATION

This is a summary of the implementation of the search engine described in the previous section. It contains original designs and codes from the author

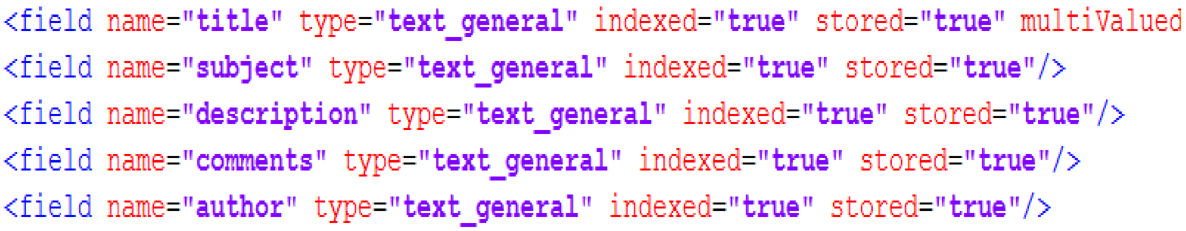
4.1. Solr Server Setup

Solr initial setup is important as it defines the system configuration and gives the blueprint to the documents that shall be indexed into the system.

4.1.1. Schema

Schema contains all definitions and details about the entities to be indexed. The crucial part of the schema is the declaration of all fields or properties of the documents, and how they should be processed at indexing and querying time. In Solr 4, it is possible to customize the type of a field and define how tokens should be filtered during index creation. This is essential in storing recency information and how to use it for ranking.

Each indexing strategy requires its own schema to employ its unique implementation. The only common features between the strategies is the default search field that contains every details about the document.



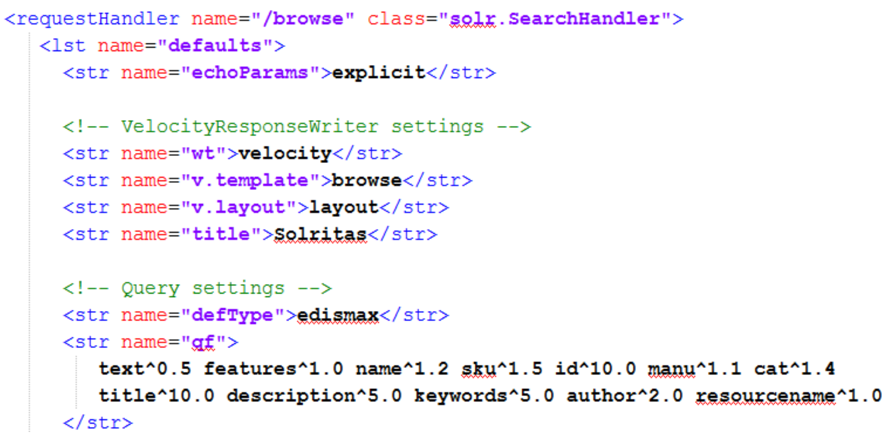
```
<field name="title" type="text_general" indexed="true" stored="true" multiValued
<field name="subject" type="text_general" indexed="true" stored="true"/>
<field name="description" type="text_general" indexed="true" stored="true"/>
<field name="comments" type="text_general" indexed="true" stored="true"/>
<field name="author" type="text_general" indexed="true" stored="true"/>
```

Figure 6: A typical design of Solr schema

4.1.2. Configuration

All parameters for automated Solr server configurations are contained in this file. The heart of this setup lies in the definition of handlers, including create, update and query requests. It is also important that an appropriate lock is defined to avoid any synchronized writing errors to the index [16].

Since the same dataset is indexed three times into different strategies, it is necessary to configure each of them on their own server, which is part of a general Solr core container.



```
<requestHandler name="/browse" class="solr.SearchHandler">
   <lst name="defaults">
     <str name="echoParams">explicit</str>

     <!-- VelocityResponseWriter settings -->
     <str name="wt">velocity</str>
     <str name="v.template">browse</str>
     <str name="v.layout">layout</str>
     <str name="title">Solritas</str>

     <!-- Query settings -->
     <str name="defType">edismax</str>
     <str name="qf">
        text^0.5 features^1.0 name^1.2 sku^1.5 id^10.0 manu^1.1 cat^1.4
        title^10.0 description^5.0 keywords^5.0 author^2.0 resourcename^1.0
     </str>
```

Figure 7: Configuration of one request handler

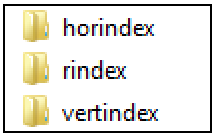
4.2. Index Time Implementation

This section describes the implementation of components that is used to index documents. The full process begins with feeding documents to the index writer. They will be analyzed with different filters before the final token stream is processed and then index is created and stored in the database.

4.2.1. Indexer

This component determines whether to create a new index or modify an existing one. The traditional Lucene index writer by default would duplicate the index and thus one document could be stored as multiple entities. In some situation, it could be beneficial to maintain multiplicity of a single entity, but in this project, this characteristic is undesired and thus, the indexer is adjusted to overwrite an existing index

Indexer is the most important component for indexing documents as it dictates the scheme to store indices. As indexer handles each document, it determines which field to create and how to store data for that field, depending on the schema.



horindex
rindex
vertindex

Figure 8: Separate storage locations for indexer

Indexer is specific for each strategy. Vertical implementation is straight forward, while the other strategies require further data processing before inserting into the indexer. Because indexer is strategy-specific, it would write indices to different storage location. This is important because it would avoid any overwritten of same entity by different indices.

4.2.2. Analyzer

This component is used to initialize an indexer. Documents are analyzed into token stream and filtered before stored into the database. This customized analyzer employs several filter functions and a random seed to generate a dummy popularity value for each document. Lucene provides a collection of filters for document analysis, most of which are often used in more complex structured entities. In this project, only three main filters are considered:

- *Whitespace Filter*: a general word delimiter that tokenizes based on white space
- *N-gram Filter (and its variant Edge N-gram Filter)*: tokenizes into contiguous subsequence of the word, this filter enable partial term query.
- *Stop-word Filter*: during tokenization, this filter ignores stop-words predefined in a list. These words are usually articles and preposition that would not make sense in tokenizing information
- *Lowercase Filter*: this filter avoid case sensitive search by storing all text values in lowercase

4.2.3. Payload Filter

Payload is an optional feature provided by Lucene. It is a metadata associated with each occurrence of a term to give the token certain level of significance. A payload is often used to store weights or other semantic information on specific terms

In this project, payload is used to store recency and popularity information of the document. Since it operates at word level, the same information is stored for every word in a document. This might seem to create unnecessary redundancies, but it could also be an advantage. Although when a document is first the entire body contain the same date and time information, when it is modified and re-indexed, only the changed text should have their new recency information. Likewise, popularity could potentially be different among the term in a document.

Unlike traditional methods in which payload data is statically defined within the document, this system dynamically generates metadata to be stored into payload. In other words, payload is only generated at indexing time. This feature allows live indexing and re-indexing of documents, which is crucial in recording time that would be used later for query-independent scoring.

Payload can be used to store any type of metadata for scoring purposes. In this project, two pieces of information, Recency and Popularity, share their spots in the payload as a byte representation of string. They are separated by a hyphen. This design is scalable, for any number of independent features.

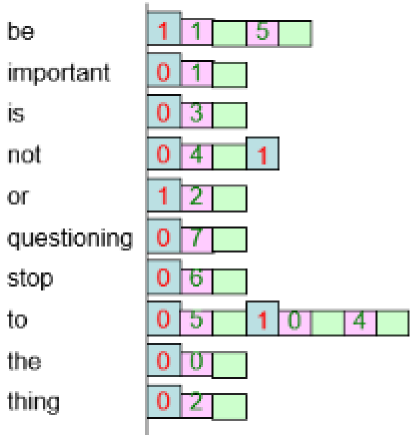


Figure 9: Example of payloads in inverted index

4.3. Search Time Implementation

This section describes the implementation of components that is used during query time. Input query is analyzed into token stream. It is then used to search for matching documents and then used by the scorer to rank result before a final list is returned to the front-end. This portion also measures retrieval time for each configuration.

4.3.1. Search Handler

After an input text is accepted by the system, it is immediately processed into tokens. These tokens are used to determine what type of query the system should handle (simple, single field…). The search handler might need to apply additional sub-queries. After that, the tokens are checked against all documents.

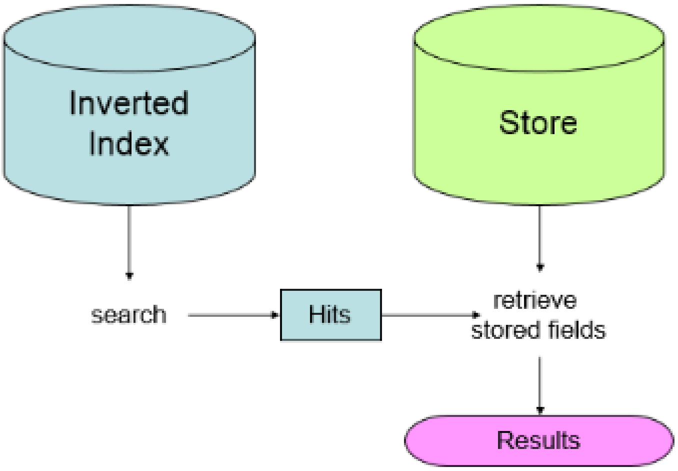


Figure 10: Workflow of a search handler

4.3.2. Scorer

A scorer calculates the final score for any entities based on query. A scorer contains all functions to compute the similarity between a query and an entity. More relevant results in higher score. In addition, the scorer also contains two discount factors, ranging from 0 to 1, to indicate the degree of recency and popularity.

In the no-scoring scenario, the scorer simply returns a constant for all computation, rendering all documents in different for any input query identical. In this case, the result would be listed in the same order the indices are stored. In the extended version of BM25, the score function is modified to retrieve and apply recency and popularity information from payload.

4.3.3. Timer

This utility measures retrieval time, denoted as the chronological difference from the moment the query is accepted to the moment the result list is returned. The timer has the uncertainty level of one hundredth of a second. Results of the timer can be used to determine the efficiency of an indexing strategy.

4.3.4. Result Formatter

Results are put into a list in order of descending score, and returned in a format that contains only necessary information such as entity properties, score and retrieval time.

5. PERFORMANCE

After the system is established in accordance of previous section, its performance is tested against different datasets

5.1. Query Type

A query, as defined previously, is an English text understandable by human. In this project, however, queries shall not follow regular grammar and punctuation. Instead, raw-text queries are strictly structured in a specific format.

5.1.1. Default Term Query

Term query accepts simple inputs, which could be a single word or a phrase of multiple words separated by whitespace. This text is searched over the default field. This makes large result pool because as long as the query terms appear in any part of the entity, it would be a match.

For queries that contain multiple terms, each would be searched separately to obtain multiple result lists. Then, they are selectively filtered, and only those that appear in all lists (entities that match all keywords) will be kept in the final result.

5.1.2. Term Query Over Single Field

Query over the default field return all entities that contain the terms, but majority of them are not really relevant. The result pool can be narrowed by giving a specific search field. This query is expected to give more desired result as it enhances accuracy by narrowing search fields. In addition, it is also expected to be faster due to much smaller search space.

A typical query of this time contains a single pair: a field and a set of terms separated by colon. Terms are whitespace separated

5.1.3. Query Over Multiple Fields

This is a free-form composition of the previous two. This type captures the most general query. A query may contain one or more pairs that are comma-separated. A pair contains 2 parts, a field and set of terms separated by colon, and field can be omitted. Set of terms are whitespace separated.

This type of query can be seen as a combination of multiple sub-queries, each of which is searched over a field. Results of each sub-query are then combined into a single list and ordered by the specified ranking function.

5.2. Medicare Helpful Contacts Dataset

Medicare Helpful Contacts (MHC) is a relatively small dataset taken from the governmental database for healthcare. The dataset is 10 megabytes in size, contains approximately 5,000 records, and with up to 15 features but only 6 useful fields are used for indexing purpose [6].

This dataset is too small such that all queries happen within a fraction of a second. Therefore, it is not objective to draw any conclusion regard efficiency and effectiveness of the engine. However, there are some interesting patterns and observations that are shown to be useful when evaluating the system.

5.2.1. Examination on Ten Queries

Ten queries have been selectively chosen (see Table 4) to account for all three query types explained in the previous section. Each of these queries is run 3 times with each of the indexing strategies for each ranking scheme. All results are recorded in Table 5, 6, and 7 to be used as reference for this examination. The retrieval time measurements are in seconds.

Table 4: List of queries for MHC Datasets

No.	Query
Q1	Hospital
Q2	medical assistance
Q3	washington health insurance
Q4	"State":California
Q5	"Organization Name":insurance program
Q6	"State":california,"Agency Name":healthcare research
Q7	"Organization Name":financing,"Agency Name":surgical facilities,"State":California
Q8	children,"State":california
Q9	cancer society, ca
Q10	"Organization Name":nurse,program

Table 5: Retrieval time with relevancy scoring for MHC

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	0.007	0.013	0.005	0.005	0.012	0.005	0.005	0.013	0.004
Q2	0.008	0.015	0.007	0.008	0.015	0.008	0.006	0.015	0.007
Q3	0.009	0.016	0.008	0.008	0.016	0.010	0.016	0.015	0.009
Q4	0.016	0.012	0.013	0.015	0.014	0.013	0.015	0.017	0.009
Q5	0.009	0.012	0.003	0.007	0.013	0.004	0.007	0.014	0.007
Q6	0.012	0.013	0.011	0.009	0.010	0.012	0.014	0.019	0.013
Q7	0.022	0.029	0.019	0.023	0.028	0.023	0.012	0.018	0.013
Q8	0.012	0.009	0.013	0.012	0.010	0.012	0.011	0.012	0.009
Q9	0.012	0.008	0.013	0.012	0.009	0.010	0.007	0.010	0.008
Q10	0.013	0.018	0.011	0.012	0.014	0.003	0.012	0.008	0.006

Table 6: Retrieval time with extra fields for MHC

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	0.018	0.017	0.018	0.017	0.026	0.015	0.012	0.016	0.017
Q2	0.019	0.026	0.019	0.018	0.028	0.014	0.010	0.018	0.018
Q3	0.020	0.021	0.014	0.019	0.025	0.019	0.017	0.018	0.016
Q4	0.016	0.019	0.013	0.016	0.020	0.015	0.013	0.019	0.014
Q5	0.017	0.019	0.013	0.016	0.016	0.012	0.017	0.017	0.013
Q6	0.025	0.023	0.017	0.024	0.018	0.017	0.015	0.016	0.023
Q7	0.023	0.021	0.018	0.023	0.020	0.019	0.018	0.021	0.022
Q8	0.014	0.022	0.011	0.013	0.021	0.013	0.016	0.021	0.013
Q9	0.013	0.019	0.012	0.014	0.019	0.013	0.013	0.010	0.013
Q10	0.013	0.018	0.013	0.013	0.019	0.014	0.013	0.010	0.013

Table 7: Retrieval time with payloads for MHC

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	0.047	0.066	0.033	0.032	0.053	0.025	0.029	0.048	0.028
Q2	0.042	0.057	0.030	0.032	0.049	0.034	0.026	0.053	0.029
Q3	0.032	0.044	0.022	0.022	0.041	0.029	0.021	0.034	0.019
Q4	0.031	0.026	0.014	0.021	0.024	0.016	0.013	0.022	0.012
Q5	0.017	0.052	0.014	0.023	0.089	0.011	0.017	0.051	0.014
Q6	0.017	0.018	0.019	0.016	0.019	0.019	0.017	0.017	0.018
Q7	0.024	0.034	0.013	0.019	0.034	0.015	0.014	0.022	0.016
Q8	0.024	0.034	0.019	0.024	0.032	0.014	0.014	0.021	0.015
Q9	0.014	0.024	0.018	0.015	0.023	0.014	0.014	0.022	0.013
Q10	0.009	0.012	0.008	0.007	0.012	0.005	0.008	0.015	0.006

Although all query runtimes are less than a tenth of a second, it can be seen that there is a clear difference in result between searching using relevancy scoring (Table A.2) and the other two schemes. This is expected because there are substantially less computations needed. Meanwhile, apart from several outliers, results for the other two scoring schemes do not show much differences in general.

Another important observation is that certain queries show favorable indexing strategy. For instance, queries over default field performs better with Reduced index for single terms, but as the number of terms increases, it favors Horizontal index. On the other hand, queries over specific field work better with Vertical index. In addition, more complex query performs better with Reduced index

5.2.2. Experiment in Depth

This section examines more closely a single query. This query is search over multiple fields and is sophisticatedly structured.

“State”:California,“Topic Name”:medicare
options,health plan choices

Figure 11: Chosen query for experiment in depth on MHC Dataset

The chosen query is generic and complex enough to represent almost any queries for this data set and thus can be used to evaluate the system performance in depth. This query has 3 parts, each of which is a featured sub-query. Part 1 is searching for a single term in a field, part 2 is searching for multiple terms in a fields, and part 3 is searching for multiple terms in general.

Below is the results returned by searching the query using BM25F scheme and Query Independent scheme. Since both solution using payloads and extra fields would yield the same result, it is only listed once in this table

Table 8: Difference in results due to scoring schemes

Result using Query Independent scheme	Result using BM25F scheme
1/Doc: 1460 … 5/Doc: 4619 … 18/Doc: 4566 19/Doc: 4568 20/Doc: 4569 … 25/Doc: 4577 … 42/Doc: 4457 43/Doc: 4458 44/Doc: 4459	1/Doc: 1460 2/Doc: 4457 3/Doc: 4458 4/Doc: 4459 … 15/Doc: 4566 16/Doc: 4568 17/Doc: 4569 … 22/Doc: 4577 … 26/Doc: 4619

The only entity remains its top position is document 1460, which has a large score margin compared to the rests. The others documents have same BM25F scores, tiebreaking by the order they are indexed. Using Query Independent scheme would adjust this score with the popularity payload (which is completely random at indexing time). In practice, payload is a much better tiebreaker than a random generator or pre-order tiebreaker because it actually gives more information about the entity.

Next, the query is run 1000 times using Query Independent scheme, each time notes the index strategy with the fastest retrieval time.

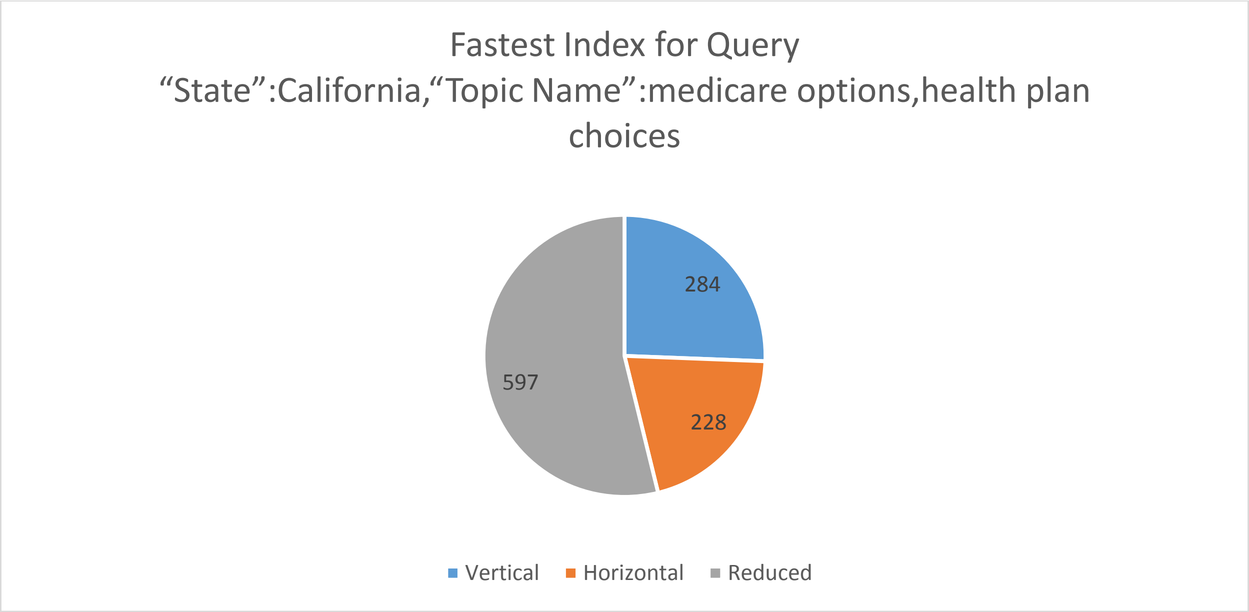


Figure 12: Indices for fastest retrieval time in MHC Dataset

**** NOTE: the sum of all counts is 1109, exceeding 1000. This is because there are occasion where more than 2 indices yields the fastest time.*

More than half of the time, Reduced index shows its superiority over the others. Vertical and horizontal index appear to be equally efficient for complex query. The distribution could have been more favorable for vertical or horizontal index if a less sophisticated query is used instead.

5.3. International Aiding Dataset

International Aiding (IA) is distributed by AidData, an online portal for information on global scale development and finance. IA contains resources of international aiding project of the modern world since 1945 with variety of finance-funded activities, include those that do not fit the ODA definition [8].

This is a large dataset with over one millions records, approximately 1GB of text. The raw data contain over 100 columns but have been processed to capture the 35 most important features as fields. Any missing fields are padded with empty strings.

This dataset is sufficiently large for testing on single node and is expected to give more realistic, reasonable and accurate results.

5.3.1. Examination on Ten Queries

Ten queries have been selectively chosen (see Table) to account for all three query types. Similar to the previous experiment, each of these queries is run 3 times with each of the indexing strategies for each ranking scheme. Table should be used as reference for this examination. The retrieval time measurements are in seconds.

Table 9: List of queries for IA Datasets

No.	Query
Q1	`united nations`
Q2	`world bank lead frank woerden`
Q3	`"donor":imf`
Q4	`"long_description":education investment`
Q5	`"year":2000,"donor":united states`
Q6	`"recipient":viet nam,"donor":thailand`
Q7	`greater mekong,"commitment_amount_currency":usd`
Q8	`"borrower":goverment,water`
Q9	`"source":website,health ministry`
Q10	`"title":goods,"plaid_sector_name":industrial development,retailing network`

Table 10: Retrieval time with relevancy scoring for IA

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	0.957	1.093	0.992	0.990	1.152	1.025	1.043	1.066	1.028
Q2	1.528	1.695	1.502	1.493	1.662	1.418	1.467	1.420	1.344
Q3	1.084	1.160	1.039	1.050	1.070	0.965	1.045	1.036	1.003
Q4	1.328	1.452	1.286	1.305	1.265	1.240	1.284	1.267	1.232
Q5	1.003	0.971	0.946	0.957	0.949	0.925	0.990	0.980	0.965
Q6	0.940	0.966	0.954	0.994	0.962	0.953	0.992	1.011	0.953
Q7	0.925	0.884	0.878	0.985	0.944	0.881	0.946	0.963	0.878
Q8	1.115	1.053	1.229	1.056	1.032	1.089	1.035	1.056	1.131
Q9	1.355	1.550	1.252	1.103	1.220	1.085	1.120	1.129	1.094
Q10	1.028	0.904	1.048	1.007	0.836	0.939	0.991	0.917	0.929

Table 11: Retrieval time with extra fields for IA

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	2.209	2.274	2.262	2.158	2.060	2.098	2.125	2.067	2.150
Q2	1.753	1.697	1.723	1.712	1.667	1.694	1.722	1.690	1.725
Q3	1.466	1.586	1.544	1.524	1.490	1.495	1.485	1.553	1.513
Q4	2.341	2.407	2.261	2.248	2.338	2.259	2.273	2.222	2.209
Q5	2.178	2.226	2.142	2.167	2.147	2.102	2.128	2.132	2.093
Q6	1.676	1.804	1.599	1.669	1.747	1.642	1.669	1.708	1.642
Q7	1.859	1.818	1.772	1.833	1.767	1.753	1.820	1.766	1.724
Q8	1.911	1.878	1.894	1.902	1.868	1.908	1.903	1.861	1.913
Q9	2.567	2.515	2.413	2.410	2.399	2.340	2.427	2.384	2.387
Q10	1.743	1.698	1.641	1.693	1.669	1.632	1.765	1.679	1.628

Table 12: Retrieval time with payloads for IA

	Measure 1			Measure 2			Measure 3		
	V	H	R	V	H	R	V	H	R
Q1	2.616	2.519	2.597	2.528	2.508	2.484	2.469	2.437	2.435
Q2	1.791	1.771	1.785	1.986	1.674	1.785	1.884	1.768	1.911
Q3	1.604	1.706	1.592	1.597	1.594	1.591	1.601	1.638	1.605
Q4	2.327	2.903	2.223	2.312	2.489	2.007	2.244	2.543	2.190
Q5	3.219	2.539	2.078	2.603	2.717	2.308	2.504	2.611	2.281
Q6	1.657	2.598	2.014	1.703	2.096	1.953	2.005	1.902	1.688
Q7	1.690	2.553	2.102	1.682	1.655	1.643	1.661	1.667	1.608
Q8	2.612	2.298	2.363	2.012	1.986	1.985	1.981	1.977	1.984
Q9	3.014	2.687	3.089	2.731	3.187	2.452	3.064	3.181	2.566
Q10	1.828	1.803	2.007	1.809	1.780	1.778	1.808	1.774	1.776

This dataset shows more realistic results. Similar patterns from the previous observation can be seen in this dataset as well. In addition, there appears to be a clear line between BM25F and Query Independent scheme, with the latter taking more time to search. This is expectable as the data volume grows, it takes more time to compute payload factor. However, the difference is not significant compared to how much No-scheme is faster than BM25F, which is almost a factor of 2. This indicates that most of computational time is used to determine query specific feature, and query-independent features only account for a small portion of query time.

Another important observation is that Vertical index efficiency drop significantly. From making around 3 fastest out of 10 queries in the small dataset, it now can only make 1 fastest time. Obviously, there are no sufficient evidence to support due to small number of measurements, but this is an indication that Vertical index might not be as efficient at this size of data

5.3.2. Experiment in Depth

Similar to the MHC Dataset, a query is selectively chosen to examine how indexing strategies behave in large database. This query is similarly structured as its MHC counterpart.

“title”:goods,“plaid_sector_name”:industrial development,retailing network

Figure 13: Chosen query for experiment in depth on IA Dataset

The query is also run 1000 times to determine which indexing strategy on average would yield the fastest query time.

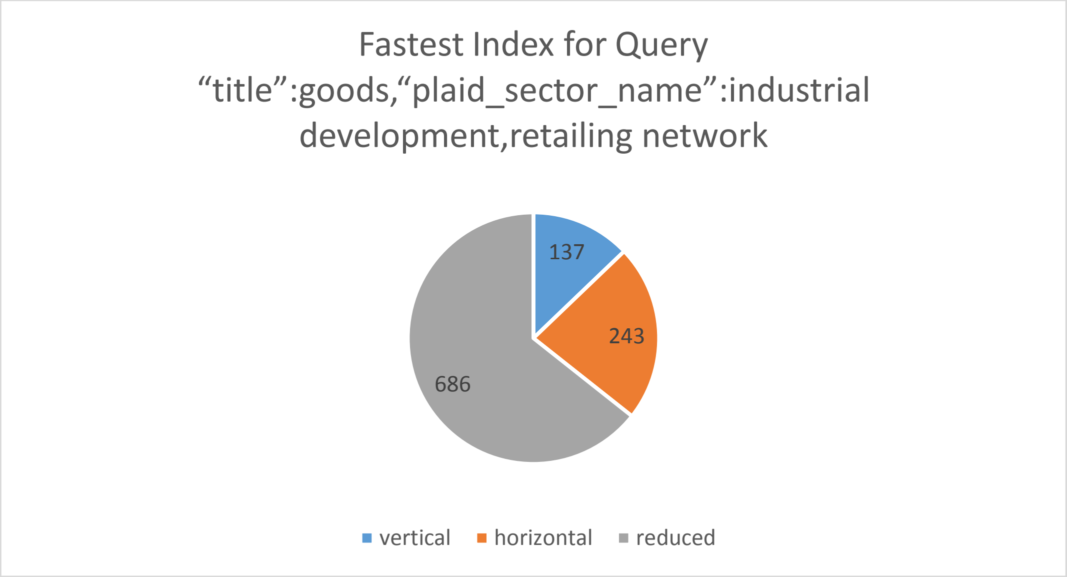


Figure 14: Indices for fastest retrieval time in IA Dataset

*** Again, there are occasion where 2 or more indices give best result*

The chart is a solid evidence showing how Reduced index becomes predominantly efficient in this large dataset. Meanwhile, Horizontal index appears indifferent and Vertical index becomes less efficient compared to their performance in small dataset

5.4. Discussion

It is clearly observed that the average time when using only relevancy score is much less than when deploying query independent features. This is expected as in the more calculations the system needs to compute, the longer it takes to retrieve information. In the third scenario, even though Recency and Popularity information have been pre-computed and stored with the payloads, the algorithm takes slightly more time. This is important because it shows that the system spends most of its time computing relevancy and only requires a small amount of time to embed more information into the ranking. Hence, it shows the prominent result of using query independent features.

This result suggests that more accurate results can obtained by injecting pre-calculated independent features and used them as either a discount factor or an additive at query time. These pieces of information do not require extra time during indexing, and their required spaces are relatively small if they are stored as payloads instead of additional storage fields.

However, there is one big limitation. Pre-calculated values cannot adjust themselves to the future changing conditions. The only feature that is self-adjusted is timestamp-based recency but this is only checked during query time. One way to resolve this is re-indexing, which allows a one-time update to every feature and their respected payloads.

5.5. Reduced Indexing Factor

This is an additive and independent experiment from the above, focusing solely on reduced indexing strategy. It is seen in the previous sections that Reduced index in the most efficient method in the long run. A question arises that how the reduced indexing factor could affect its performance. Consider these three scenarios for different level of importance in Reduced index:

Table 13: Level of importance in Reduced index

3 levels	4 levels	5 levels
• Important • Neutral • Unimportant	• Very Important • Important • Neutral • Unimportant	• Very Important • Important • Neutral • Unimportant • Useless

Supposed there is a known way to classify all fields equally into these categories in Reduced index. Then, running the same 10 queries above for both dataset for 100 times would give the following average results:

Table 14: Average measurements of querying time with different levels of importance in Reduced index for both MHC and IA dataset

	Medicare Helpful Contacts			International Aiding		
Level	3	4	5	3	4	5
Q1	0.025	0.028	0.037	2.597	2.540	2.520
Q2	0.034	0.030	0.032	1.785	1.763	1.770
Q3	0.029	0.027	0.032	1.592	1.612	1.480
Q4	0.016	0.020	0.021	2.223	2.320	2.016
Q5	0.011	0.014	0.017	2.078	2.197	1.923
Q6	0.019	0.021	0.021	2.014	1.963	1.952
Q7	0.015	0.016	0.020	2.102	2.059	2.036
Q8	0.014	0.016	0.016	2.363	2.161	2.264
Q9	0.014	0.015	0.017	3.089	2.949	2.756
Q10	0.005	0.009	0.009	2.007	2.101	1.957

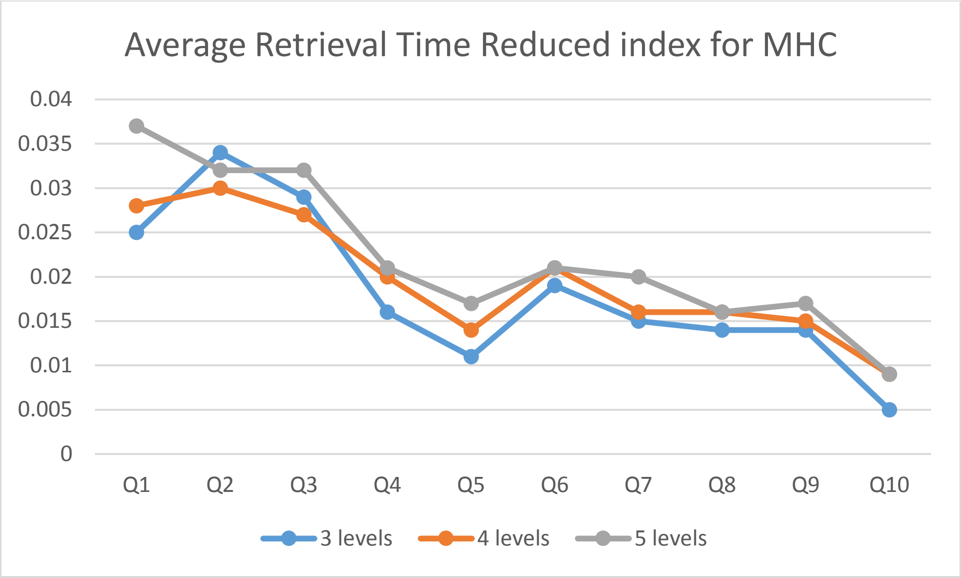


Figure 15: Average retrieval time for different reduced indexing factors in MHC

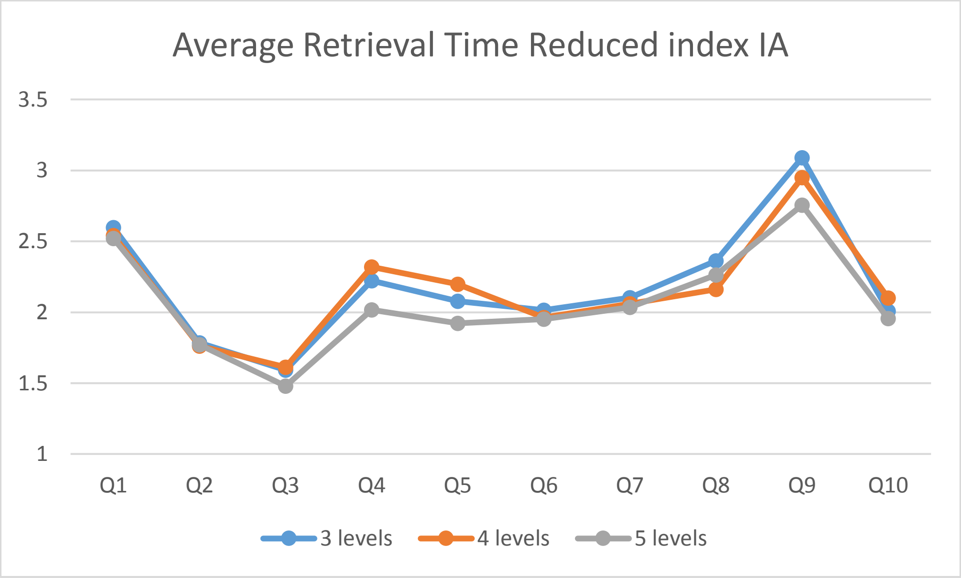


Figure 16: Average retrieval time for different reduced indexing factors in IA

There are no clear distinctions between three lines in both graphs, indicating that number of level divisions does not affect much to the outcome. However, there are some interesting patterns can be observed from the graphs:

- In MHC graph (smaller set with fewer fields), 3-level model tends to perform better than 5-level
- In IA graph (larger set with more fields), 5-level performs slightly better
- 4-level model in both graphs appears to swing slightly between the other models. Its average performance is somewhat in between
- The gaps between the lines in IA is less than in MHC, and the lines in IA form similar shapes

Some of these observations are explainable. Since fields are randomly and equally distributed into 3, 4, and 5 categories, the number of information contain in each category become N/3, N/4 and N/5 respectively (with N is the number of fields for a document). MHC has only 6 indexed fields and thus N/4 and N/5 make it behaves like a vertical index. Meanwhile, IA has 30 indexed fields, and thus using 3, 4, and 5 levels would yields 10, 8, and 6 fields each, and thus 5 levels with 6 fields each would be more balanced in term of creating and searching an index.

Therefore, the more balanced it is during indexing, the faster the average retrieval time will be. As a result, larger set of fields means more categories in Reduced index to balance the load. However, because these categories are also used to measure level of importance for each field, it does not make sense if there are too many of them. Indeed, the fewer the number of fields, the less storage for payload information needed, which explain why horizontal index tends to outperform vertical in simple query with multiple terms. Overall, this result gives an insight of how this factor can affect performance based on dataset volume. However, discussion of how to obtain the optimized Reduced index factor is out of scope for this paper.

6. CONCLUSION

This project implements a Lucene-Solr full-text based search engine with three different indexing strategies. Although their performances are very similar, it is suggests that Reduced index is a better choice, for its flexibility in the number of categories. There are no predominant strategy in general, but instead, index design should be based on the volume (number of records) and the cardinality (number of fields) of the data. Vertical index may work best for system that mainly supports simple query. Meanwhile, if a system is expected to receive complex query then horizontal and Reduced index will be a better choice, given that the number of fields is reasonably small. In practice, Reduced index uses categories for implementing level of importance, and thus the load those may not be as equal as it is in this project. Nevertheless, the number of categories for Reduced index should be engine-oriented or dataset-specific to ensure highest performance; and it should be limited to avoid becoming vertical index. Results from this project also promote the use of query independent features in ranking. With small tradeoff time margins, using these features to enhance accuracy is more effective than the similarity functions themselves. There are many way to store and use these features, but using payloads has been shown by this project to be among the most effective, and another alternative is using extra fields.

This project could be further developed with other aspect of entity search such as query suggestion and recommendation. There are projects on this topic that use similar technology, which could potentially be integrated with the solution proposed in this paper.

REFERENCES

[1] Aguera, Jose. Arroyo, Javier. “Using BM25F for Semantic Search” *Proceedings of the 3rd International Semantic Search* Workshop. ACM Press, New York, USA, 2010

[2] Blanco, Roi. Mika, Peter. Vigna, Sebastiano. “Effective and Efficient Entity Search in RDF Data” *International Semantic Web Conference (ISWC 2011)*, Germany, 23-27 October 2011. p83-97

[3] Catena, Matteo. Macdonald, Craig. “On Inverted Index Compression for Search Engine Efficiency”. *Advances in Information Retrieval*. Gran Sasso Science Institute. Volume 8416, 2014, pp 359-371

[4] Dali, Lorand. Fortuna, Blaz. Tran, Thanh. Mladenic, Dunja. “Query-Independent Learning to Rank for RDF Entity Search” *Extended Semantic Web Conference (ESWC 2012)*, Greece, 27-31 May 2012. p484-498

[5] Fontoura, Marcus. Gurevich, Maxim. “Efficiently Encoding Term Co-occurrences in Inverted Index” *Proceedings of the 20th ACM international conference on Information and knowledge management* ACM Press, New York, USA, 2011. P307-316

[6] Government Medicare Data. Internet. data.medicare.gov/data/medicare-s-helpful-contacts. 10 Janurary 2015.

[7] Gupta, Vineet. “Indian Union Budget 2015 – The Other Perspective”. Internet. vineetguptablog.wordpress.com 6 March 2015

[8] International AidData. Data Hub. Internet. datahub.io/dataset/aiddata. 15 March 2015

[9] Josifovski, Vanja. “Comparison of Similarity Search Algorithms Over Inverted Indexes”. Stanford University. 27 September 2010

[10] Miller, Mark. “Query Parsing Tips & Tricks in Solr”. ApacheCon Europe 2012. Rhein-Neckar-Arena, Sinsheim, Germany. 5-8 November 2012.

[11] MIMIC II: Clinical Database Overview. Internet. www.physionet.org/mimic2. 2 April 2015

[12] Peng, Jie. Ounis, Iadh. “Selective Application of Query Independent Features in Web Information Retrieval”. Glassglow, United Kingdom. 2009

[13] P´erez-Ag¨uera, J.R., Arroyo, J., Greenberg, J., Iglesias, J.P., Fresno, V. “Using BM25F for semantic search.” *International Semantic Search Workshop* (2010). ACM Press, New York, USA, 2010. p1-8

[14] Potter, Timothy. “Boosting documents by Recency, Popularity, and User Preferences”. San Francisco. 25 May 2011

[15] Mika, Peter. “Distributed Indexing for Semantic Search”. *International Semantic Search Workshop* (2010). ACM, 2010. p1-4

[16] Solr Tutorial. Internet. cwiki.apache.org. 20 December 2014

[17] Vercoustre, Anne-Marie. Thom, James. Pehcevski, Jovan. “Entity Ranking in Wikipedia” *ACM Symposium on Applied Computing* (2008). ACM Press, New York, USA, 2008. p1101-1106