CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Algorithm Selection for Document Retrieval

How Algorithm Selection can Reduce Energy Consumption in Search Engines by Improving Query Latency

Master's thesis in Computer Science and Engineering

Albin Pansell
Simon Riis

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Algorithm Selection for Document Retrieval

How Algorithm Selection can Reduce Energy Consumption in Search Engines by Improving Query Latency

Albin Pansell
Simon Riis

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Algorithm Selection for Document Retrieval
How Algorithm Selection can Reduce Energy Consumption in Search Engines by Improving Query Latency
Albin Pansell
Simon Riis

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Oskar Hagman, Vesiro
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Algorithm Selection for Document Retrieval
How Algorithm Selection can Reduce Energy Consumption in Search Engines by Improving Query Latency
Albin Pansell
Simon Riis
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Energy consumption in data centers is increasing and is expected to use a significant share of the world's energy in the future. As the quantities of data are also increasing, searching this data uses substantial computational resources. One way to reduce the energy consumption of search clusters is to reduce the search latency. The type of search used for unstructured data, such as in web search engines, is called *top-k document retrieval*. Several different top-k document retrieval algorithms exist, but no single algorithm performs best in all cases. A technique that can be used in problems with this characteristic is *algorithm selection*. Algorithm selection is a technique that involves predicting and using one, out of a set of complementary algorithms, that will perform the best in a specific problem instance. In this thesis, we study whether algorithm selection is a viable approach for improving query latency, over state-of-the-art query processing algorithms. To do this, we first measured the performance of a set of relevant state-of-the-art top-k document retrieval algorithms in different queries. Using this performance data, we trained a machine learning classifier to predict which algorithm would perform the best, given a query. We show that the potential latency improvement of the algorithm selection approach varies with the database size. The largest potential improvement we observed is near 25%. We present an algorithm selector which achieves 17 out of the 25%. We also discuss how this could reduce the energy consumption of the querying process by around 15%, but also some practical considerations.

Keywords: information retrieval, top-k document retrieval, algorithm selection, query processing, energy consumption, machine learning, random forest, decision tree

Acknowledgements

We thank Oskar Hagman and Oscar Widén at Vesiro for giving us the opportunity to write this thesis. Special thanks to Oskar Hagman for taking time out of his busy day to serve as our supervisor and provide invaluable guidance. We also thank Felix Naredi and Felix Korch for giving us technical advice and support throughout the work. We thank Pedro Petersen Moura Trancoso for answering all our questions and giving us feedback during the writing of this thesis. Last but not least, we thank Fredrik Hamrefors and Adam Törnkvist for their great feedback on our writing.

Albin Pansell and Simon Riis, Gothenburg, 2024-06-27

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ANOVA	Analysis of Variance
BMM	Block-Max Maxscore
BMS	Block-Max Score
BMW	Block-Max WAND
DaaT	Document at a Time
ML	Machine Learning
PISA	Performant Indexes and Search for Academia
PLL	Postings List Length
SBS	Single Best Solver
SIMD	Single Instruction Multiple Data
TaaT	Term at a Time
TMS	Term-Max Score
VBS	Virtual Best Solver
VBMM	Variable Block-Max Maxscore
VBMW	Variable Block-Max WAND

Contents

List of Figures

List of Tables

1

Introduction

Energy consumption in data centers is continuously increasing. When search engines process millions of queries per day a substantial amount of computational resources are consumed. This becomes more and more relevant as the amount of searchable data grows, further accelerating the trend of increasing energy consumption in search engines [1]. Real-world applications affected by this trend are web search engines and database management systems for large-scale data analysis, like Elasticsearch. If optimizations can be made to improve latency when processing queries in these systems, less energy is consumed [2]. In this thesis, we aim to improve query latency for full-text search engines.

The problem of retrieving the top k best matching documents from a dataset, given a search query, is referred to as top-k document retrieval. A query processing algorithm performs the retrieval given a set of documents, a query, and a value for k. The set of documents is usually represented by an index data structure. A particular instance of these presumptions and input values, used by a query processing algorithm, is referred to as a search instance throughout the thesis. A top-k document retrieval algorithm then works by searching through the index for the query terms, applying a ranking function to score how relevant each document is in relation to the query. Finally, the k documents with the highest scores are returned by the query processing algorithm.

The naive way of performing query processing is exhaustive search, where the index is fully traversed linearly. Two query processing algorithms, WAND [3] and Maxscore [4], have come to play a large role in the field of top-k document retrieval, as they support skipping documents in the index. Several optimizations have been made on top of these algorithms. Although, the latest algorithms do not necessarily perform best in all scenarios. A study providing experimental results on multiple query processing algorithms and multiple index compression techniques shows that the algorithm with the lowest latency can be different for different query lengths [5]. Crane et al. [6] also suggest that no single algorithm performs best in all cases of the top-k document retrieval problem.

Approaches to exploit algorithm complementarity in problems of this sort, where the best algorithm varies depending on the problem instance, have been studied and are referred to as *algorithm selection*. In algorithm selection, the goal is to select the

best out of a set of algorithms depending on the instance of a problem.

In this thesis, we design an algorithm selector that uses Machine Learning (ML) to predict the optimal algorithm to use for each incoming query. To accomplish this we first benchmark a set of the most relevant query processing algorithms on a wide range of queries. The benchmarking is performed over data collections of different sizes to show how the approach scales with database size. The benchmarking data is then used to train an ML classifier to perform the algorithm selection.

Finally, we propose an algorithm selector for top-k document retrieval, consisting of a decision tree model basing its decisions on query characteristics. This selector yields a speedup of up to 17.7% over the Maxscore algorithm. Furthermore, this leads to 15.0% potential reduction in energy consumption.

1.1 Aim

From a broad perspective, our goal is to reduce energy consumption by applying optimizations that lower the latency of query processing. Our work aims to study whether algorithm selection is a viable approach for improving query latency, over state-of-the-art query processing algorithms. Our thesis explores this by stating the following research questions.

- **RQ1 - Complementarity between algorithms**
  Do existing query processing algorithms complement each other in a way that speedup can be achieved by selecting the optimal algorithm in each search instance?

- **RQ2 - ML model mapping**
  Is it possible to map search instances to optimal algorithms using a simple ML model, based on the available features?

- **RQ3 - Robustness to change in index size**
  Can an algorithm selection model trained on an index with a specific collection size (uncompressed sizes of indexed data) generalize to work well on other indexes of other collection sizes?

1.2 Delimitations

In this master thesis, we have delimited our work to build upon query processing algorithms processing queries in a safe, early termination fashion. Early termination, refers to techniques for avoiding exhaustive index traversal, and safe, meaning producing the exact same top k results, as an exhaustive search, for a particular search

instance.

An inverted index with documents can be traversed using different approaches. The two most common approaches are document-at-a-time (DaaT) and term-at-a-time (TaaT). If the DaaT approach is used, lookups are performed for all terms in a query, document by document. If the TaaT approach is used, the whole index is traversed for every term. As many early termination techniques have been proposed for DaaT, we have decided to limit our work to finding optimizations within this approach.

Another delimitation is the choice of document datasets to investigate. We have focused our work on segments of web data from the Common Crawl dataset. Section 3.2.2 further describes the selection of datasets. Future work could include investigating this approach for data from other sources and more topic-specific datasets.

1.3 Ethical Considerations

For this master thesis, ethical considerations involve the risks of applying the emerged research on search systems of large scale. If algorithms proposed by this thesis were to be implemented on large-scale systems and the decision of an algorithm selector was not based on sufficient data, it could potentially lead to sub-optimal algorithm selections being made. As the processing of search queries can be critical in many applications this could potentially have bad implications on latency and functionality of large-scale systems. Malfunctioning systems could in the worst case lead to unethical consequences, such as strategic decisions being made on incomplete data.

If the proposed algorithm selector were to be implemented without severe bugs that would change the set of presented documents, the risk of negative effects is likely small. This is because the algorithm selector and existing algorithms should be functionally equivalent while differing in implementation. Thus, they should behave the same and the only observable difference would be the query latency.

Another ethical consideration is for whom the research would provide value. Potentially, the research would mostly benefit large established corporations that could incorporate it in their data centers. However, decreasing the world's energy consumption would still be a common benefit.

2

Theory

In this chapter, we first present general aspects of top-k document retrieval, in section 2.1. Next, the background regarding the creation of indexes is explained in section 2.2. In section 2.3, query processing algorithms are listed and described. Strategies for index traversal and early termination are also presented. Section 2.4 provides a background on the field of Algorithm Selection.

2.1 Top-k Document Retrieval

The problem of retrieving the top k best matching documents from a dataset, given a search query, is referred to as top-k document retrieval. Given a set of documents, an index is created as a data structure to organize the documents in an easily searchable way. A query processing algorithm for top-k document retrieval works by searching through the documents, and applying a scoring function to score how relevant each document is in relation to the query. Finally, the k documents with the highest scores are returned.

Organizing the documents in an index allows relevant documents to be found fast, given a query of terms. Indexes are usually also compressed to consume less disk space and to fit in memory. The most common type of index for top-k document retrieval systems, Inverted index, is explained in section 2.2.

Query processing is the core part of top-k document retrieval and can be classified in multiple ways. This includes different traversal strategies, safe or unsafe processing, early termination approaches, which ranking function is applied, and finally, which query processing algorithm is used. Aspects of query processing are further explained in section 2.3.

2.2 Inverted Index

In information retrieval systems document indexes are used to look up documents when a search query is processed. Inverted indexes are a common approach where documents are indexed on a term-by-term basis [7]. Similarly to an index at the end of a book, for each term in a dataset, the index stores a list of document IDs containing that term. The lists of document IDs are called *postings lists* and the

entries are called *postings* [7]. The most basic form of posting contains just the document id, but can also contain other information such as the frequency of the term in the document. Note that the documents themselves are not stored in the index.

Figure 2.1 shows how postings are structured in an inverted index. The postings lists are organized in a directory structure called a *vocabulary*. The vocabulary stores an entry for each term, containing the number of documents where the term appears, i.e. the *postings list length* (PLL), and a pointer to the start of the postings list [8]. The term entries may contain additional information such as pre-computed maximum scores per term, so-called term-max scores (TMS), that algorithms can use to process queries faster. This will be explained further in section 2.3.5.

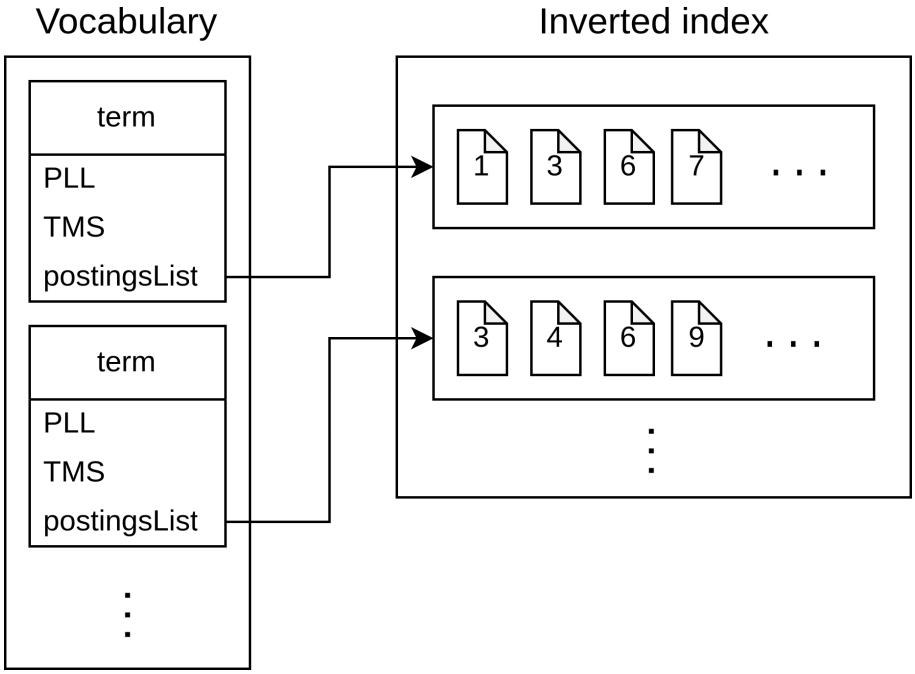

Figure 2.1: Inverted index

2.2.1 Index Compression

Index compression is an important factor in document retrieval. Compressing the index allows for it to consume less space on disk, but also allows for larger parts of the index to be kept at higher levels of the memory hierarchy. Ideally, the entire index could be kept in main memory. Because of this, index compression algorithms must not only be space efficient, but also inexpensive to decompress.

A basic way to compress an index is to use *d-gaps* [9]. D-gaps refer to the differences between consecutive document IDs in a postings list. Document IDs can be

very large, possibly ranging from zero to billions. Therefore a way to compress an index is to store only the differences between document IDs instead of the absolute values. These values will be significantly smaller, and require much fewer bits. Compression algorithms may then be further applied to the resulting sequence of integers.

Postings lists are often split into blocks of 64 or 128 documents, which are compressed and decompressed separately. Advanced query processing algorithms can avoid decompression of entire blocks while traversing the postings lists. Following are brief descriptions of a selection of commonly used compression algorithms.

Vbyte

Variable Byte Encoding (Vbyte) [10] (also referred to as Varint) is a compression algorithm that stores integers in a variable number of bytes by allocating one bit at the end of each byte in the binary representation to indicate whether or not it is the last byte of the integer. While this can lead to less efficient representation it still benefits overall from the efficient representations of small values.

Varint-G8IU

Modern compression algorithms utilize Single Instruction Multiple Data (SIMD) instructions to offer shorter effective decompression speed through parallel decoding [11]. SIMD refers to the type of parallel processing where the same operation is executed on multiple data points simultaneously. SIMD processing units can also be referred to as array processors [12]. Varint-G8IU [11] is a SIMD version of the Vbyte encoding, which stores as many integers as possible into eight consecutive bytes together with one descriptor byte. The descriptor byte contains the number of compressed integers in the eight bytes. A single SIMD instruction can then decode all of the integers.

SIMD-BP128

SIMD-BP128 [13] is another SIMD-based compression. SIMD-BP128 implements a vectorized binary packing over blocks of 128 integers. It stores these integers into as few 128-bit words as possible and the blocks are then grouped into groups of 16 blocks, preceded by a 128-bit descriptor. A single SIMD instruction can decode 16 128-bit words simultaneously.

QMX

The QMX encoding [14] builds upon SIMD-BP128, but instead of the two-way split between descriptor and payload, it is split into three parts: payload (Quantities), run length (or Multipliers), and descriptor (eXtractor). In QMX the descriptor uses only one byte, the remaining bits are used to encode run-lengths. Either the payload or the descriptor can be run-length encoded. QMX has been shown to be more space-effective and more efficient than SIMD-BP128 and Varint-G8IU [14].

2.2.2 Document Reordering

The space efficiency of any d-gap-based index compression ultimately depends on the sizes of the differences between the document IDs. These in turn depend on the document IDs themselves. By default, documents are assigned IDs based on the order they are parsed into the index. But because of the great impact document ID assignment has on compression, techniques have been studied to reassign these IDs to decrease the d-gaps and achieve better compression. To achieve this, documents containing a specific term should have IDs that are close to each other, and overall documents with similar contents should have similar document IDs [15].

Document reordering techniques can also be applied to improve the query latency when traversing an index. The goal is to reorder documents in a way that minimizes the d-gaps but also increases the probability of finding matching documents earlier in the index resulting in a shorter lookup time. The top k documents can be retrieved quicker if they are clustered and large parts of the index can be skipped when traversing.

A common way to reorder documents is reordering by a specific document feature, e.g. URL of an online document [16]. This is done due to the assumption that documents with similar URLs have similar content and thus will result in postings lists with smaller d-gaps and better compress rates.

One state-of-the-art document reordering technique is recursive graph bisection [17]. This technique recognizes that graph reordering, a technique that is effective at increasing the locality of the representations of graphs and compression of social networks, can be applied to improve compression in inverted indexes. In particular, the algorithm uses graph reordering to optimize the size of the graph compressed using d-gap encoding.

2.3 Query Processing

Processing a query involves searching through the postings lists of an inverted index. Query processing algorithms do this by moving pointers over the postings lists of the terms contained in the query, in a coordinated fashion [18]. If the query is a ranked query, such as in top-k document retrieval, the documents that satisfy the boolean properties of the query, should also be ranked according to some ranking function.

2.3.1 Traversal Strategies

The two main ways of searching an index are called document-at-a-time (DaaT) and term-at-a-time (TaaT). In the DaaT approach, the postings lists are traversed concurrently, evaluating an entire document across all postings lists where it occurs. The pointers are then incremented to evaluate the next document [18].

In TaaT processing, entire postings lists are processed one at a time. This means that it is necessary to keep some data structure to accumulate the partial document scores of each document [18]. This is because the final document scores are not known until all postings lists have been traversed. This is a significant disadvantage of TaaT processing.

2.3.2 Ranking Functions

In order to select the top k documents, the documents need to be ranked according to some score value. The score of a document is calculated by applying a ranking function. BM25 [19] is a commonly used ranking function, and is a simple but effective ranker for processing bag-of-words queries. It uses the number of occurrences of the words, and the length of a document to score the relevance of the document.

2.3.3 Safe vs Unsafe Query Processing

Retrieving the top k documents can be done in either a safe or unsafe fashion. A safe query processing algorithm is a deterministic process and ensures that the same k documents are returned as if an exhaustive search was performed. Unsafe query processing enables possible further improvements in performance by relaxing the requirement of returning exactly the same k as an exhaustive search.

2.3.4 Early Termination

The naive implementations of DaaT and TaaT must access every posting of every query term. In DaaT every candidate document must be scored, and in TaaT a score contribution must be computed for each posting [18].

When using exhaustive traversal algorithms, the length of postings lists becomes a major bottleneck. In general, postings lists grow linearly with the size of the data and can grow up to hundreds of gigabytes for frequent terms [20]. Early termination is an important technique to reduce this problem.

If it is not required to score all documents, and it is enough to return the top k documents, it is possible to skip full evaluation for parts of postings lists. This works as long as no possible top-k candidates are skipped. This approach is the most common one for early termination in safe query processing algorithms [3]. The specific ways how this is handled will be explained further in section 2.3.5 below.

2.3.5 Query Processing Algorithms

Multiple query processing algorithms have been designed in the field of top-k document retrieval. Here, algorithms are presented ranging from pure exhaustive index traversal to more complex approaches utilizing early termination techniques and indexes split into blocks. All algorithms presented follow the common DaaT traversal strategy, described in section 2.3.1.

Exhaustive DaaT

Exhaustive DaaT is the basic or "naive" implementation of the DaaT index traversal strategy, which fully scores every document in the query term postings lists. The algorithm traverses the index by keeping one pointer for every postings list, i.e. every term. First, the documents are sorted in increasing order of document ID [21]. At the start of index traversal the pointers are set at the beginning of each postings lists as shown in Figure 2.2 (a). The document with the lowest ID among the pointers is scored, in this example document 1. After that, this pointer is incremented, as shown in Figure 2.2 (b). The process is repeated by scoring the new document with the lowest ID, document 3. Once again the corresponding pointers are incrementing, shown in Figure 2.2 (c).

A min-heap of size k is allocated to store the current top k documents [21]. If a document scores higher than the current minimum document in the heap, it is inserted and the previous minimum is removed if the heap is full. The algorithm continues to traverse the documents in the index until the pointers have exceeded the postings lists.

On top of this algorithm, different skipping techniques can be applied to enhance the pointer movement. This can lead to chunks of documents being skipped and a decreased latency of the whole index traversal. Such skipping techniques are shown when presenting the algorithms below.

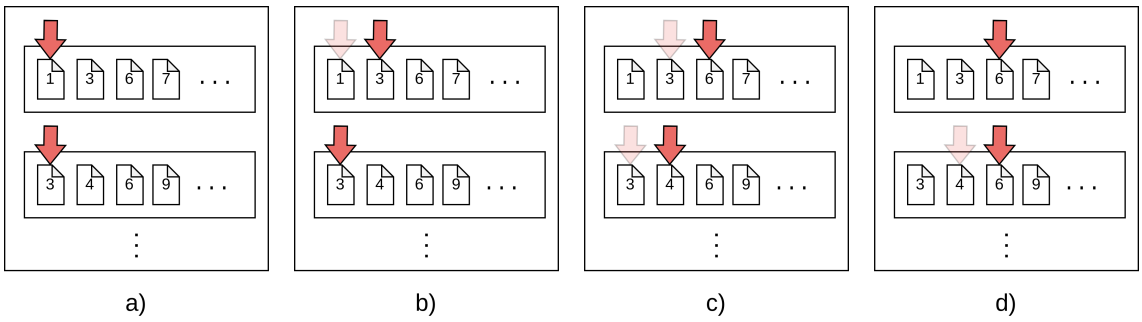

Figure 2.2: Pointer movement in the inverted index

WAND

The WAND (Weak AND) algorithm is an early termination algorithm that uses a preliminary evaluation to determine candidate documents for full evaluation. Only candidate documents are fully evaluated, meaning scored with the scoring function. The preliminary evaluation uses query-independent term contribution scores, to establish an upper bound for the document scores, which can be used to determine whether or not a document could place among the top k documents [3]. These scores are referred to as term-max scores, TMS, mentioned in section 2.2. If the upper

bound of a document's score is below the current top-k threshold, the score of the kth highest scoring document, the document will be skipped. If it is higher, meaning that it might place in the current top k, the document is fully evaluated to determine whether this is the case. As the algorithm progresses, the threshold increases. As more high-scoring documents are added to the heap, more documents can be skipped.

The TMSs are computed at indexation time by traversing the postings lists of each term and for each document computing the frequency of the term in the document, divided by the document length. The maximum value of all the documents is the maximal contribution of a term to any document score. The inverted index is then augmented by storing the TMS of each term in the vocabulary of the inverted index [3].

The algorithm starts by sorting the query terms in increasing order of the document ID of their pointers. Then, it computes a pivot term, by accumulating the TMSs in the sorted order until the sum exceeds the threshold. The document pointed to by the pivot term pointer is the minimum document ID that could be a scoring candidate. This document is called the pivot document. Once the pivot document has been identified, the algorithm checks if the first pointer in the sorted order points to the same document as the pivot document. If this is the case, the document is scored. Otherwise, the algorithm tries to advance the pointers between the first and the pivot term to the pivot document ID. This advancement is done using an index skipping mechanism which can save index accessing time, due to the postings lists being sorted. After the pointers have been moved, the algorithm sorts the terms again and repeats this process until a suitable pivot term no longer can be identified.

An example of the skipping mechanism described above is shown in Figure 2.3. Here the TMSs of the first three postings lists sum to 7.2. This is greater than the current threshold of 6.5. Therefore, the third postings list is the pivot. Because the pointer of the first postings list does not point to the same document ID as the pointer of the pivot, the pointers of the first and second postings lists can be skipped forward to the same document as the pivot. Thus, the algorithm avoids scoring the documents that are skipped.

Block-Max WAND

The Block-Max WAND (BMW) algorithm is based on the WAND approach outlined above, with one key addition: the introduction of an augmented inverted index structure called a block-max index [20]. A major disadvantage of the WAND algorithm is that TMS scores are global maximums for entire postings lists. Therefore, a large outlier will set the TMS score of a postings list to a large value, which will cause the algorithm to significantly overestimate the relevance of many of the documents in the list. The idea behind the block-max index is to store a maximum contribution score for each block of a compressed index, called a block-max score (BMS). These scores enable more accurate preliminary document scores and therefore skipping of

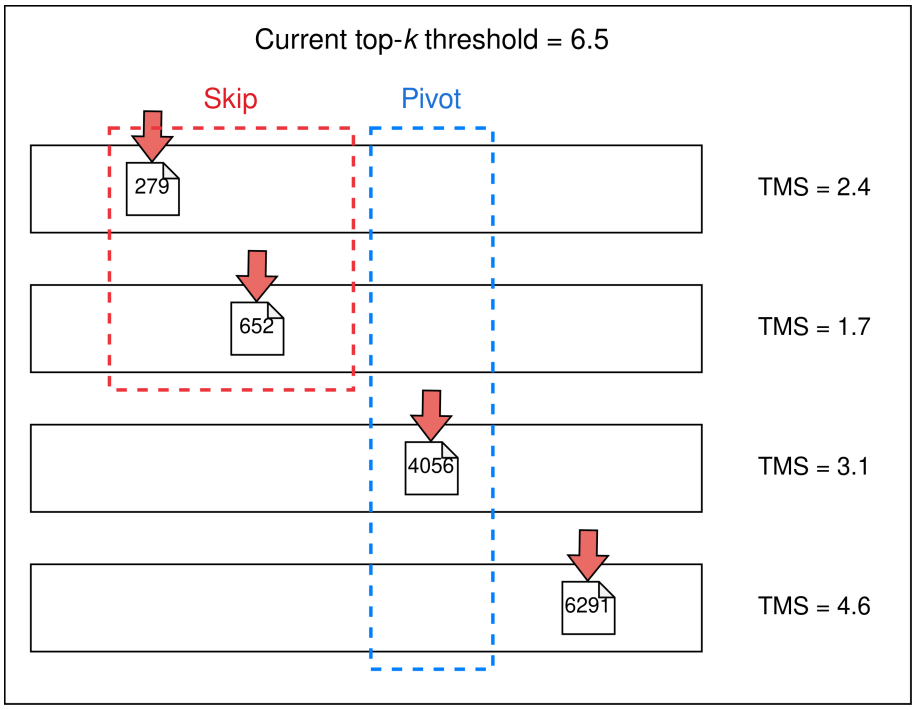

Figure 2.3: An example showing the WAND skipping technique

more documents.

While inverted index blocks are usually compressed, the BMSs are often stored separately and uncompressed. The fact that there are compressed blocks with uncompressed BMSs gives rise to two types of pointer movement: deep pointer movement and shallow pointer movement. Deep pointer movement involves decompression of a block, while shallow pointer movement moves a pointer without decompressing the corresponding block, thanks to the uncompressed BMS.

Because shallow pointer movement does not need to decompress the block, it is used whenever possible. Before BMW evaluates a document, it will utilize the BMS, through shallow pointer movement, to see whether or not to check if the documents in the block have to be evaluated. If they cannot make it into the top k results, the pointer is moved forward at least beyond the end of the current block. If it cannot be skipped, deep pointer movement will need to be used to evaluate each document in the block.

Variable Block-Max WAND

Variable Block-Max WAND (VBMW) is the BMW algorithm with support for variable block partitioning. It aims to further improve the BMS used in BMW. In the BMW algorithm, the performance is dependent on the block size. If the block size is too large, the probability of having a large outlier in the block is higher; and if the block size is too small, skips will be less effective.

VBMW uses variable-sized blocks to adapt to the distribution of scores in postings lists by taking regularities and variances of the scores into consideration [22]. It defines *block error* to be the sum of differences between a document's actual score and the maximum score of its corresponding block. The goal is then to find a block partitioning that minimizes the block error, by viewing the problem as an unconstrained shortest path problem. To solve this an approximately optimal partitioning strategy is introduced, which gives an approximately optimal solution in a reasonable time.

When creating a variable block partitioning the algorithm needs an input parameter for performing so-called Lagrangian relaxation of the unconstrained shortest path problem. Lagrangian relaxation is a method for relaxing mathematical optimization problems to turn them into simpler, approximate problems. This input parameter is referred to as *lambda* and stands for the fixed cost that is added to every edge in the graph of this shortest path problem. Different values for lambda will result in different numbers of edges in the shortest path, and in more concrete terms: different sizes of the blocks in the partitioning. If a certain block size is desired, the appropriate lambda value needs to be searched for experimentally.

Maxscore

The Maxscore algorithm is another optimization of the DaaT strategy [4]. Just like the WAND algorithm, Maxscore uses TMSs stored in the inverted index to skip parts of the index. The main idea behind the Maxscore algorithm is to use the TMSs and the current threshold value to identify which set of terms must be present in a document for it to be a potential top-k candidate [18]. Just like in WAND, the threshold progressively increases, leading to more documents being skipped.

The Maxscore algorithm starts by sorting the pointers in order of the size of their postings lists [18]. Unlike WAND, Maxscore only sorts the pointers once, at the beginning of processing. Before the top-k heap has been filled, Maxscore works like Exhaustive DaaT. Once it has been filled, just like WAND, Maxscore uses the score of the lowest-scoring document in the heap as the threshold when preliminary evaluating new documents. Documents with a preliminary score below the threshold are skipped.

After sorting the pointers, the algorithm splits pointers into two groups; the required set and the optional set [18]. For a document to be a possible candidate, it must include at least one of the terms from the required set. This split is computed by iterating over the pointers and accumulating their TMSs until the sum exceeds the current threshold. This means that the split has to be recomputed each time a document is added to the heap, since the threshold changes.

Once the split has been identified, Maxscore uses Exhaustive DaaT over the required set. Once a candidate document is identified, the pointers in the optional set must be moved to the ID of the identified document to score it.

Block-Max Maxscore

Block-Max Maxscore (BMM) is an extension of the Maxscore algorithm to use the same augmented Block-Max index as BMW [23]. BMM adds an estimating step before scoring a candidate document selected from the required term set. This step calculates the document's upper bound score using the Block-Max score. If this upper bound score does not exceed the top-k heap threshold, the algorithm will skip to the next validated candidate document.

Variable Block-Max Maxscore

Variable Block-Max Maxscore (VBMM) is the BMM algorithm with support for variable block partitioning. VBMM has the same implementation and goal as VBMW, as described above. It enables a variable and optimized block partitioning by taking the distribution of scores in postings lists into account.

2.4 Algorithm Selection

It is commonly observed that, for a specific computational problem where a large variety of high-performance algorithms exist, there is no single algorithm that is best in all instances of the problem. Rather, different algorithms perform best in different instances, which gives rise to a phenomenon called *performance complementarity* [24]. Automated algorithm selection provides an approach to exploit this performance complementarity.

There are several different approaches to algorithm selection, but the specific problem relevant to this thesis is referred to as *per-instance algorithm selection*. The per-instance algorithm selection problem can be defined as the following, as described in Kerschke et al. [24]:

> Given a set I of instances of a problem P, a set $A = A_1, ...A_n$ of algorithms for P and a metric $m : A \times I \to \mathbb{R}$ that measures the performance of any algorithm $A_j \in A$ on instance set I, construct a selector S that maps any problem instance $i \in I$ to an algorithm $S(i) \in A$ such that the overall performance of S on I is optimal according to metric m.

2.4.1 Virtual and Single Best Solvers

The goal of an algorithm selector is to achieve performance as close as possible to a hypothetical optimal selector on the set of instances. This hypothetical best selector is referred to as the *virtual best solver (VBS)*. The VBS provides an upper bound for what performance is achievable with algorithm selection. On the other hand, there is the concept called *single best solver (SBS)*. SBS refers to the single algorithm in A that is best, on average, over all instances. This provides a lower bound to the performance of a reasonable algorithm selector. If it performs worse than the SBS,

the algorithm selector is of no use.

The difference or ratio between VBS and SBS is referred to as the *VBS-SBS gap*. The VBS-SBS gap is useful for reasoning about what performance gains are achievable through per-instance algorithm selection. It can be used to analyze a computational problem to see whether the algorithm selection method can be effective. If A demonstrates high complementarity between algorithms, this gap will be large, i.e. the possible performance gains are high.

Finally, the ratio of the VBS-SBS gap that a given algorithm selector can close, provides a metric for its performance. Together the VBS, SBS, VBS-SBS gap, and ratio form a useful set of metrics to analyze the potential and performance of an algorithm selector.

2.5 Machine Learning

Supervised ML is a paradigm in ML that focuses on modeling the relationship between some input variables $\mathbf{x}$ and their corresponding output variable y [25]. The input variables are referred to as *features*. For all non-trivial ML problems, this relationship between input and output is difficult or impossible to describe explicitly or mathematically.

In supervised ML a distinction is made between whether the output variable is numerical or categorical. In the numerical case, it is called *regression* and in the categorical case, it is called *classification*. This thesis concerns itself with classification. In classification, the goal is to predict or classify the categorical *class* to which an instance belongs based on its input feature data. The output variable in this case is referred to as class or *label*.

To model the relationship between input and output, some model or algorithm is trained on some training data containing pairs of the input features and their corresponding class, which is known in advance. It is very important that after training, the model can generalize to new instances that it was not trained on and accurately predict their class. Because of this, ML datasets are often split into a training set and a test set. The training data is used to train the model and the test data is then used to evaluate the model on data that it has not seen before, to test that it can generalize.

In cases when the classes of a dataset are imbalanced, or different misclassifications have different costs, instance weights can be applied. This involves assigning a weight to each instance, and the model will prioritize instances with higher weights during training.

Most ML algorithms involve some sort of settings or parameters that control the complexity of the model or some details of the learning process. Because these parameters often cannot be learned they are referred to as *hyper-parameters*, as opposed to just parameters, which typically refer to the learned parameters of a

model. Because hyper-parameters cannot be learned by the model, and are often hard to set intuitively, they are often optimized using some hyper-parameter optimization method. The simplest form of hyper-parameter optimization is *grid search*. Grid search is essentially an exhaustive search for the best-performing configuration in the grid of hyper-parameter values.

2.5.1 Machine Learning for Algorithm Selection

For an algorithm selector to map available features to the most suitable algorithm, an ML model is usually developed. There are different approaches for this, but the simplest is to train a classification model, with the available algorithms as classes. The best-performing algorithm becomes the label for each set of instance features [26]. For algorithm selection in the field of top-k document retrieval, the latency of the ML model needs to be very low, due to query processing algorithms operating at latencies below milliseconds. This leads to low-latency algorithms such as decision trees and random forests being suitable.

Figure 2.4 shows a schematic overview of the algorithm selector development process. First, the problem is characterized and a set of algorithms and features are identified. A set of relevant problem instances is also defined. This should be as representative as possible of the problem. The algorithms are then benchmarked on the problem instances, and the features in the feature set are computed for the instances. This results in feature data and performance data. This data then serves as training data for an ML model. The ML model is the core of the algorithm selector. Once it has been trained it can be incorporated into an algorithm selector, which can be evaluated. The algorithm selector performance data can then be used to go back and tune the hyper-parameters of the ML model, or remove a subset of the features.

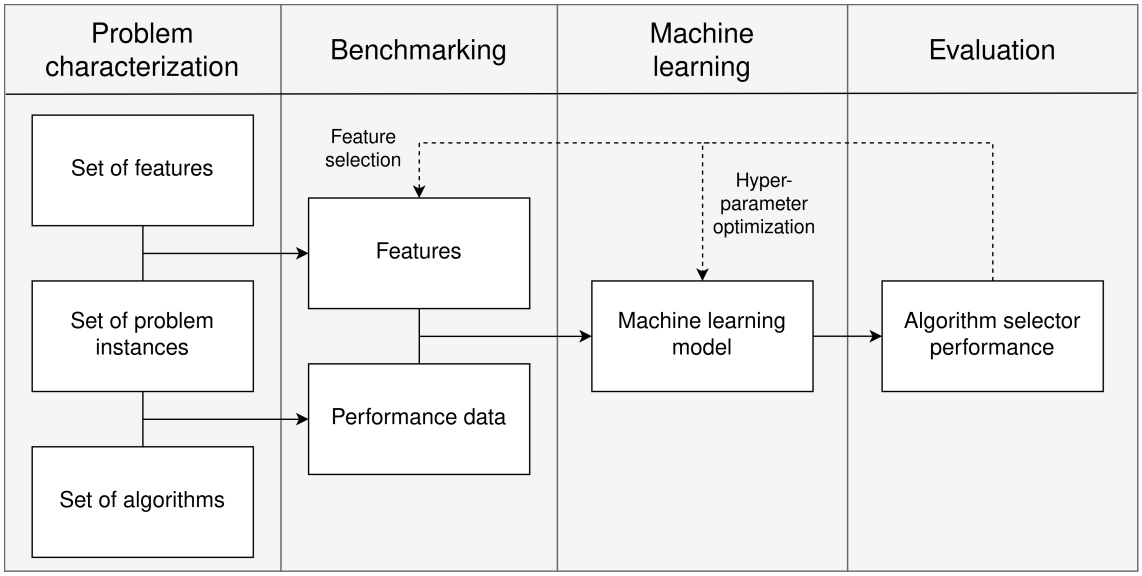

Figure 2.4: Schematic of the algorithm selector development process

2.5.2 Features for Algorithm Selection

One of the most important considerations in developing an effective algorithm selector is to have informative and cheaply computable problem instance features, $\mathbf{f}(i) = [f_1(i), ..., f_k(i)]$ for a given problem instance i [24]. This is important because the algorithm selector needs to be able to learn a suitable mapping from problem instances to the most suitable algorithms. An ideal feature should have the following five properties: *informative*, *cheaply computable*, *complementary*, *generally applicable*, and *interpretable*. It is chiefly important that features are informative in a way that they allow for distinction to be made between problem instances. However, for the advantages gained by algorithm selection to not be outweighed by the cost of computing the features, they need to be cheap to compute. They should also be complementary since it is computationally wasteful to compute redundant features and in addition, redundant features can be problematic for ML. Features should also be generally applicable so that they can be used to map a broad range of instances to algorithms. Lastly, features should ideally be interpretable to give insights into instance properties.

2.5.3 Analysis Techniques

To analyze the behavior and performance of an ML model and the underlying problem, statistical analysis methods are often used. If a classification model misclassifies instances, a *confusion matrix*, also known as error matrix, is commonly used [25]. The confusion matrix of a classification problem with N classes is an $N \times N$ matrix with predicted class on the x-axis and true class on the y-axis. The elements of the matrix are the number of instances that fall into each combination of true and predicted class. Thus, if a perfect model is analyzed, all non-zero elements will be on the diagonal (all instances have the same predicted class as their true class). The confusion matrix makes it possible to analyze which classes are misclassified and which classes are confused with each other. For reference, the confusion matrix containing algorithm selector predictions for feature analysis is shown in section 4.4.3, Figure 4.11.

Analyzing the features and labels of an ML problem with statistical methods can give insights into the problem. Feature correlation analysis is one such technique. By looking at the correlation between two features it is possible to investigate whether some features may be redundant [25]. If two features have a very high correlation, it suggests overlapping or redundant information between the features, and either could possibly be removed. To study pairwise feature correlation, a correlation matrix can be produced. This contains an element for each pair of features, containing their correlation coefficient. There are several different methods of calculating the correlation coefficient. A common model is the *Pearson correlation* [27]. The Pearson correlation coefficient is the ratio between the covariance and the product of the standard deviations of two variables. The result is a normalized correlation coefficient on a scale from -1 to 1 showing to what degree a pair of features correlate linearly. 1

means full positive linear correlation, -1 means full negative linear correlation, and 0 means no correlation.

Feature Importance is another important tool in analyzing the features of an ML problem. Feature importance assigns an importance score to each feature. The importance scores can then be used to compare how important each feature is for prediction. Because of this, feature importance is often used for *feature selection*, which refers to excluding some number of redundant or less important features [28]. There are several different methods for calculating feature importance. Some are model-dependent and some are model-agnostic. One model-agnostic way of calculating feature importance is to use the ANOVA (Analysis of Variance, a collection of statistical models that use variance to describe relationships between variables) F-statistic. The ANOVA F-statistic measures the variation of features in relation to different labels. Features with significant variance across the different labels have high F-statistic values and are considered important.

2.5.4 Decision Tree Classifier

A decision tree is a classifier that predicts the label associated with an instance, based on a set of features, by traveling down a tree of decision nodes [29]. The algorithm travels from the root to a leaf, and the leaf corresponds to the predicted label. Each node contains a decision on a feature, which splits the input space. For example, a node could split the input space by checking whether the term count feature is smaller or greater than 3.

There are different algorithms for training decision trees but most are based on the same fundamental strategy. The algorithms use a criterion that measures the quality of the split of a node. A commonly used criterion is the Gini impurity [30]. The algorithm starts with a single leaf node. The leaf is then split iteratively, choosing the splits that maximize the splitting criterion. Finally, the tree is pruned, as the resulting tree often becomes large. The pruning is often implemented as a bottom-up walk through the tree, replacing nodes with either a sub-tree or a leaf, based on some metric.

2.5.5 Random Forest Classifier

Random forests are ensembles, or collections, of decision trees [29]. In a random forest, each decision tree is trained on a sample of the training data, and the best split at each node is based on a random subset of features. During classification the trees in the forest vote for a class, resulting in the class with the most votes being the predicted class. Alternatively, a tree in the random forest can produce a probabilistic prediction, resulting in the predicted class being the averaged prediction of all trees. The randomness of random forests enables the models to cancel out the high variance and overfitting of their diverse individual trees.

3

Methods

To create an algorithm selector, benchmarking is needed for recording data on how well different query processing algorithms perform. First, an index was configured. Then, benchmarks were run over a selected parameter space. Next, we identified ML features in the benchmark data and analyzed the quality of these. The data was then used to train ML models to predict which algorithm would perform the best on a specific search instance. We optimized hyper-parameters for the models, implemented them as algorithm selectors in C++, and integrated them into the search engine. Finally, we evaluated the algorithm selectors against state-of-the-art query processing algorithms, to compare their latency and speedup.

In this chapter, we first show the steps of configuring our index in section 3.1. In section 3.2 we then show presumptions for benchmarking and the selection of different parameters. Next, we describe the process of creating ML models for the algorithm selectors, in section 3.3. Finally, in section 3.4, we describe the evaluation process of an algorithm selector.

3.1 Index Configuration

Before running benchmarks, an index configuration is needed. We have focused on selecting a state-of-the-art index configuration as far as possible, to evaluate the algorithm selection approach in the most realistic scenario. We selected QMX as index compression. As described in section 2.2.1, QMX is more efficient and space effective than the SIMD-BP128 and Varint-G8IU algorithm, and ultimately the best performing algorithm among the examined index compression algorithms. For the document ordering, we used the state-of-the-art technique, Recursive graph bisection, described in section 2.2.2.

Two different approaches can be taken in the case when algorithms utilize a block-divided index: fixed block size or variable block size. Due to the VBMW algorithm, presented in [22], showing a performance increment over BMW, we selected the variable block size approach. This led to VBMW and VBMM being used instead of BMW and BMM.

For the variable block size partitioning, a parameter referred to as lambda value needs to be selected. The lambda value indicates the granularity of blocks in the

inverted index, described in section 2.3. Smaller lambda values lead to smaller average block sizes. This parameter was selected by running benchmarks on the VBMW and VBMM algorithms over multiple lambda values within a common range. The experiment was performed for the collection sizes 128 GiB, 256 GiB, and 512 GiB. In the research paper proposing the variable block size approach [22], lambda values are calculated for a set of desired block sizes. The presented lambda values range from 12.0 to 64.5 for different datasets and desired block sizes. The performance of the VBMW algorithm is also shown to be better for smaller average block sizes, and therefore smaller lambda values. Based on this, the following lambda values were selected for benchmarking: 12.0, 18.0, 24.0, 35.0, 46.0.

3.2 Benchmarking

To create a basis for designing an algorithm selector, we benchmarked a set of query processing algorithms. This produced data on what algorithms performed best in what search instances. The benchmarking output data was first used to calculate metrics to show the potential speedup of the algorithm selection approach. The data was then used for training and testing an algorithm selector (ML algorithm), to obtain the final speedup for this approach.

Benchmarking was done by iterating over a parameter space, and processing queries for each parameter set. For every query processed, the parameter set was recorded together with the resulting latency, measured in μs. Characteristics of each query, PLLs, and TMSs were also recorded. These were interesting to record as they could potentially act as ML features for an algorithm selector to base its decisions on.

The algorithms used as candidates for algorithm selection are listed in section 3.2.1. The dataset and queries that were used are described in section 3.2.2. In section 4.1 the experimental setup is shown. Finally, the full set of parameter values is summarised in section 3.2.3.

3.2.1 Algorithm Candidates

The following set of query processing algorithms were selected as candidates for the algorithm selector and included in the benchmarking process:

- Exhaustive DaaT
- WAND
- Maxscore
- VBMW
- VBMM

Exhaustive DaaT is the naive approach of traversing the full index. In general, this

approach would appear to be slower but one benefit is that it does not introduce any extra algorithmic overhead. For that reason, it is still a relevant candidate. The WAND and Maxscore algorithms and the approach of introducing Block-Max indexes are prevalent in the field. Both the native algorithms and their Block-Max versions BMW and BMM have shown performance increment over Exhaustive DaaT [20], and are therefore relevant as candidates. As described in section 3.1, the variable block size implementation is preferred leading to VBMW and VBMM being used instead of BMW and BMM.

3.2.2 Dataset and Queries

The dataset we used to provide documents for indexation is called Common Crawl[1]. It is a free and open-source repository for web crawl data. It contains over 250 billion pages and has been cited by over 10,000 research papers. It consists of unstructured text data, where the top-k document retrieval problem is applicable. Furthermore, web search engines search this type of data, using the same document retrieval techniques as discussed in this thesis. This makes the Common Crawl dataset realistic and suitable for this benchmarking.

A Common Crawl archive contains around 100 TiB of compressed data. To achieve data sizes that are feasible to process, we took randomized samples from the November/December 2023 crawl archive[2]. Samples were taken of different sizes to see how our approach scaled with increasing dataset size. These are referred to as collection sizes and are set to 128 GiB, 256 GiB, 512 GiB, 1024 GiB, 2048 GiB, and 4096 GiB (uncompressed text data).

The queries used when running the benchmarks are taken from the TREC Million Query Track 2009[3], consisting of 40000 queries. Queries containing terms that are not present in our index were filtered out. This way, only terms that appeared in at least one of the documents were included and used in the benchmarking. Figure 3.1 shows the number of queries categorized by term count in the queries. This is the resulting query distribution after filtering was done with respect to the index with a 2048 GiB collection size. This corresponds to an average of 2.54 terms per query. The same filtering process was performed over indexes of all collection sizes.

K-value

The value k is the number of documents the query processing algorithm should return. This affects the query latency and is therefore varied among the most commonly used values, in our case: 10, 100, 1000. These values have been used for benchmarking in multiple research papers within the top-k document retrieval field [20] [5] [31].

[1] https://commoncrawl.org/
[2] https://data.commoncrawl.org/crawl-data/CC-MAIN-2023-50/index.html
[3] https://trec.nist.gov/data/million.query09.html

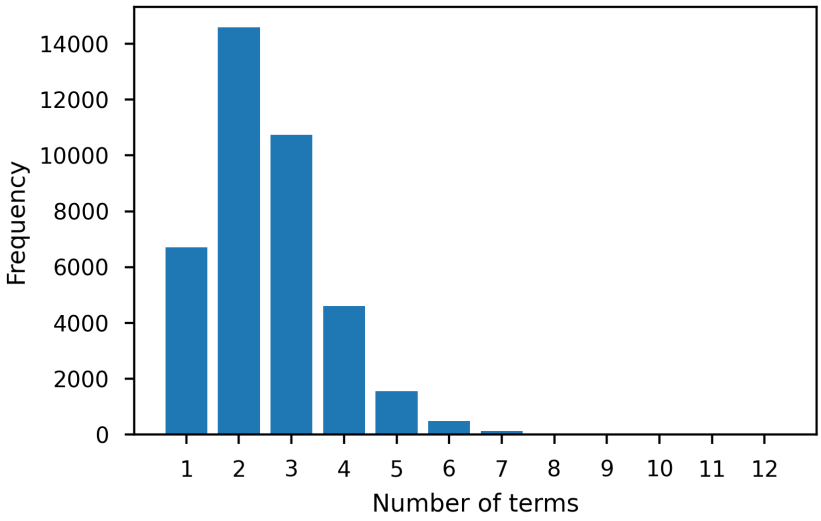

Figure 3.1: Query length distribution of the TREC Million Query Track 2009 dataset

Ranking Function

The ranking function used is BM25 due to being widely used in information retrieval systems in the industry[4]. It is also one of the most common ranking functions in the information retrieval field and is commonly used as a baseline [32].

3.2.3 Parameter Summary

A summary of parameters and their values used in the benchmarking process is shown in table 3.1 below. The selected parameters for the index configuration are also presented here.

Table 3.1: Benchmark parameter summary

Parameter	**Value(s)**
Algorithm candidates	Exhaustive DaaT, WAND, VBMW, Maxscore, VBMM
k-values	10, 100, 1000
Ranking function	BM25
Dataset	Common Crawl November/December 2023
Collection sizes (GiB)	128, 256, 512, 1024, 2048, 4096
Query dataset	TREC Million Query Track 2009
Compression	QMX
Document order	Recursive graph bisection
Lambda-values	12.0, 18.0, 24.0, 35.0, 46.0

[4]https://www.elastic.co/blog/elasticsearch-5-0-0-released#search

3.3 Machine Learning

In this section, we present the process of developing a classification model that, based on a set of query instance features, predicts the right label, i.e. the best-performing algorithm. To do this we used the data from the benchmarking as training data. Each benchmarked query is seen as an instance of the ML problem and the query processing algorithm with the best latency on that particular query is set to be the label of that instance. The features used to map the instances to the right algorithms are query-specific details, such as the term count, and characteristics of the particular postings lists of the terms in the query.

First, in section 3.3.1, we describe what ML features we identified in the benchmarking data and other features created based on the features directly present in the data. In section 3.3.2 we describe how we analyzed the quality of the features. Lastly, in sections 3.3.3 and 3.3.4 we describe how we developed and prototyped the classifiers in Python, and how we implemented them in C++ to be evaluated in the PISA search engine.

3.3.1 Features

The set of features that are made available to an algorithm selector is of critical importance. These features need to be available at query processing time and fulfill the criteria described in 2.5.2 to the largest extent possible. Primarily, they should be informative and cheap to compute.

In this case, the available features are: first, the value of k, which is directly accessible since it is needed for the query algorithm to know how many results to present. Its value has been shown to affect which algorithm is the most suitable [5], and can therefore be considered an informative feature.

The second available feature is the *term count* in the query. Query processing algorithms have access to this value implicitly. Either by counting the terms in the query or by accessing a size variable directly. This depends on the implementation. If the terms are counted, the access time becomes linear to the term count itself, otherwise the access time is constant. This feature has also been shown to affect the choice of best algorithm [5]. For the query dataset selected in section 3.2.2, the term count takes a value between 1 and 12. This means that linear time complexities, with respect to the term count, still result in short execution time, due to small term counts.

Next, the PLLs can be used as features. The PLLs are accessible through the vocabulary, as described in section 2.2. Thus, the PLLs are accessible in constant time. As there is one postings list per term, the selected query dataset results in 12 PLL features, since 12 is the longest query length. The PLLs express how long the postings lists the algorithms have to traverse are, and different algorithms might be optimal for different lengths. Thus, it is reasonable to expect that this is an informative feature.

Another feature that can be used is the TMSs. These are used by all of the algorithms, except Exhaustive DaaT, and are stored in the vocabulary, accessible at constant time. The TMSs provide new information to the algorithm selector as they contain information that affects the early termination mechanisms of the algorithms. Potentially, different values of these can be associated with which algorithm will perform the best.

Additionally, we created summary features based on the features mentioned above. We calculated the following summary features based on the PLLs and TMSs: average, median, sum, maximum, and minimum. The complexity of calculating these is linear to the term count.

3.3.2 Feature Quality Analysis

As mentioned in section 2.5.2, the features should be informative, cheaply computable, complementary, generally applicable, and interpretable. The following is a description of how we analyzed how well the features fulfilled these criteria. Regarding the interpretability, no statistical analysis was performed. We considered the term count, value of k, and PLLs to be interpretable as they are defined as real counts of terms and documents. The TMSs were considered less interpretable as this is a metric built for query processing algorithms to use during index traversal. Although, this was not enough cause to omit the TMSs as features.

Feature Informativeness

To analyze how informative our features are, meaning whether they allow for distinction between query instances, we studied how useful the features are for ML algorithms to predict the target algorithms. The first step in determining whether this is the case was to look at the achievable accuracy when fitting an ML model to the data. If a high accuracy can be achieved, it means that the set of features is informative. For this, we used a random forest classifier. A random forest model was trained without any complexity restrictions, to focus on whether the features can be used to distinctively map the queries to the right algorithm.

To further analyze the individual features and their informativeness, we used feature importance analysis. The feature importances were calculated using the ANOVA F-statistic, described in section 2.5.3.

Feature Complementarity

When analyzing the complementarity between the features, we analyzed how redundant they were. If two features essentially encapsulate the same information, it is redundant to compute and train a model on both. We used feature correlation analysis to investigate this. We computed the pairwise Pearson correlation between all features and displayed the results in a correlation matrix, described in section 2.5.3.

Feature Computation Cost

On top of looking at the algorithmic complexity of computing the features, discussed in section 3.3.1, we also measured the time it took to compute them. Retrieving single-value features (value of k and term count) takes a very short amount of time. Therefore, we focused on measuring the feature computation time of the PLL and TMS summary features. The summary features that were calculated are sum, max, and mean. To measure both how long each of these, and all together, take to compute, measurements were conducted for the following three sets of features:

1. Single-value features and PLL summary features
2. Single-value features and TMS summary features
3. Single-value features and all summary features

The summary features for PLLs and TMSs were calculated by looping over the number of terms in the query once, accumulating the sum, and checking for max. The mean was then calculated using the sum.

Feature Applicability

To analyze how generally applicable the features are, meaning how well they can enable an ML model to map a broad range of search instances to algorithms, we analyzed miss-predicted instances of the model. If there were miss-predicted instances we could look for patterns or specific kinds of instances that the features were not able to consistently map. We used a confusion matrix to visualize which algorithms were misclassified.

3.3.3 Classifier Development

When developing the classification model for the algorithm selector it is important to optimize, not directly for classification accuracy, but for the resulting latency of its selections together with its classification latency overhead. Therefore, the classifiers were hyper-parameter optimized with regard to their calculated overall query latency. This was calculated by look-ups into the benchmarking results, and adding the prediction latency of the specific model to each selection.

Two main approaches were taken during classifier development. First a random forest model, as a more complex approach, with the possibility to be implemented using generated C++ code. We used the random forest implementation of scikit-learn 1.4.2 [33], which was later translated into C++ code, as described in section 3.3.4 below. This approach is referred to as the Library implementation. The hyper-parameter grid searched for the Library implementation is shown in table 3.2. Due to very long compilation times the maximum tree depth was limited to 8, as shown in the table.

The second approach consists of a decision tree, a simpler model suitable for manual implementation using if-statements. This approach is referred to as the If-statement

implementation. Its hyper-parameter grid, which only contains maximum depth, is shown in table 3.3 for completeness. Due to fewer implementation constraints, the maximum depth was only limited to 20. This approach was modeled without including the prediction latency overhead because we assumed it to be negligible due to the low execution time of if-statements.

Before a classifier was trained on the data the feature values of the individual PLLs and TMSs were shuffled. If the PLLs and TMSs are considered as one feature each, the majority of them are often empty, as few queries contain the maximum number of terms. Shuffling is also important to not embed any specific ordering or importance of specific postings lists. The following summary metrics were also calculated based on the PLL and TMS features: average, sum, maximum, minimum, and median. The final result of the hyper-parameter optimization is one best ML model for each collection size.

Table 3.2: Hyper-parameter grid for random forest model: Library implementation

Parameter	**Values**
Maximum tree depth	1, 2, 4, 6, 8
Number of trees	1, 2, 4, 8, 16, 32, 64

Table 3.3: Hyper-parameter grid for decision tree model: If-statement implementation

Parameter	**Values**
Maximum tree depth	1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20

3.3.4 Classifier Implementation

For the Library implementation, the python package TL2cgen[5] was used. This package converts tree-based models, such as random forests, trained in Python into C++ code.

The If-statement approach uses a simple tree of if-else statements and returns the predicted label as an integer in each leaf. This implementation supports only decision trees and not random forests. To further optimize the implementation we used the compiler built-in function `__builtin_expect` to optimize the branch prediction performance. Whether to expect true or false in the nodes, is based on which child node checked the most training samples, i.e. which path is the most common.

[5]https://tl2cgen.readthedocs.io/en/latest/

3.4 Evaluation

To evaluate the algorithm selection approach, the best model for each collection size was implemented into the PISA search engine. The models that previously had been trained and tested in Python, were now translated into C++ code. This was done to measure the performance of the actual implementations, in a real search engine environment.

When evaluating the candidate models, the algorithm selectors were implemented as a step before query processing, which calculates the used features and then performs classification with these using the model. The resulting algorithm label from the classification is used to select which algorithm to run. Each candidate model was evaluated against the state-of-the-art query processing algorithms Maxscore and VBMW, by again benchmarking, as described in section 3.2.

3.4.1 Robustness to Change in Index Size

As the collection size affects a trained model through PLLs, the performance of a model might change if the index grows or shrinks. It is therefore interesting to evaluate how a model trained on benchmark data from a certain size would perform if a large number of documents were added or removed from the index. Our approach for evaluating this was to evaluate the selectors, trained on the different collection sizes, on all the other collection sizes, that they were not trained on. By doing this we gained knowledge of the models' ability to handle scaling in size of the index and indicated how much the index would be able to grow before the model would need to be re-trained.

4

Results

In this section, we present the results of the thesis. First, in section 4.1 we present the experimental setup. Next, in section 4.2, we show the results of an experiment conducted to determine a suitable value for the lambda parameter, a parameter that needs to be configured to use variable block partitioning. After this, in section 4.3, we present the results of measuring the VBS-SBS gap for different collection sizes. This is a measure of the maximum potential speedup of an algorithm selector. Then, we present the results of the feature quality analysis in section 4.4. In section 4.5 the results of the classifier development are shown. Then, we present the results of the final evaluation of the developed algorithm selector in section 4.6, which is followed by energy consumption aspects in section 4.7. Lastly, the results are discussed in section 4.8

4.1 Experimental Setup

This section outlines the experimental setup used to benchmark and evaluate query processing algorithms. The experiments are performed using a text search engine called PISA: Performant Indexes and Search for Academia [34].

4.1.1 PISA

Performant Indexes and Search for Academia (PISA) is the text search engine framework used in this work to test and evaluate state-of-the-art query processing algorithms and indexes. PISA is an open source library[1] and aims to bring together academic groups to define a reusable framework for running and reproducing information retrieval experiments [34]. It includes implementations of a set of algorithms and indexing techniques, making it easy to perform experiments. Furthermore, it aims to be extensible, making it easy to incorporate and test new algorithms. PISA is especially intended for research focusing on efficiency. Therefore, it is implemented in C++ and makes use of optimizations such as SIMD instructions, branch prediction hinting, and CPU intrinsics.

Branch prediction is a CPU functionality where CPU instructions are loaded beforehand based on which code path, branch, the execution is most likely to take. On

[1] https://github.com/pisa-engine/pisa

average this speeds up execution time. Branch prediction hinting is the technique of giving the compiler hints on what branch predictions to incorporate into the machine code. CPU intrinsics refer to built-in functions that instruct how code implementation is handled by the compiler.

4.1.2 Computer System

The experiments were conducted on a machine with a 6-core Intel Core i5-12500, 32 GiB RAM, running Linux 6.1.0. The CPU has a base frequency of 3.00 GHz and a maximum turbo frequency of 4.60 GHz. It has the following cache sizes: L1 Instruction: 32KB, L1 Data: 48KB, L2: 1280KB, and L3 (shared): 18MB. It supports the Intel AVX2 SIMD instruction set extensions.

4.2 Index Configuration: Selection of Lambda Values

On top of the selected index configuration described in section 3.1, a lambda value needs to be selected for the variable block size partitioning. The selection of this parameter is important to give the VBMW and VBMM algorithms optimal performance.

The following set of lambda values were selected to be evaluated: 12.0, 18.0, 24.0, 35.0, 46.0. The value 12.0 was found to be the optimal one among the searched values, as shown in Figure 4.1 and Figure 4.2. The figures show the average latency for different lambda values for the VBMW and VBMM algorithms. Although the value of lambda did not significantly affect the performance of VBMM, the value was selected considering both VBMW and VBMM, as it needs to be chosen at the scope of the index. Even though this is the lowest value in the span we did not find it reasonable to test even lower values, as this lambda value resulted in an average block size of around 16 documents. A too low block size introduces more overhead for handling blocks and BMSs and therefore we do not consider it feasible with lower lambda values than 12.0.

When comparing the results from the different collection sizes, we see the same pattern. This indicates that the choice of lambda value generalizes to different collection sizes, and the same value can be chosen for each. With these results, the lambda value was set to 12.0 for all experiments.

4.3 VBS-SBS Gap

In this section, we present the potential achievable speedup for the algorithm selection approach, referred to as the VBS-SBS gap. Here we answer RQ1 - Complementarity between algorithms, by showing that a VBS-SBS gap exists between them, leading to potential speedup if an algorithm selector were to select the optimal algorithm for each search instance. The benchmark results produced by the setup, described in

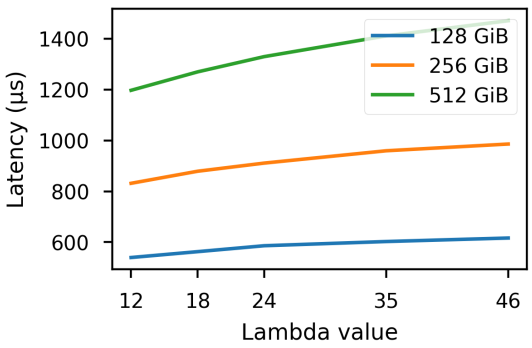

Figure 4.1: Average latency over lambda values of the VBMW algorithm

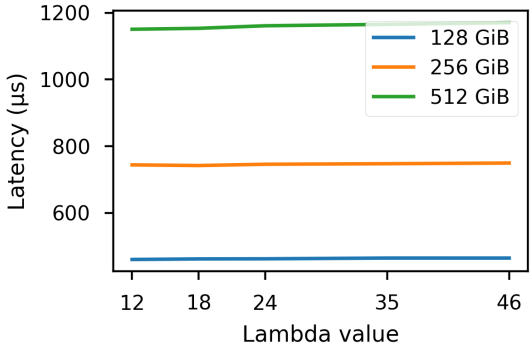

Figure 4.2: Average latency over lambda values of the VBMM algorithm

section 3.2, are used to calculate the average latencies of the VBS and SBS. For the collection sizes 128 GiB, 256 GiB, 512 GiB, and 1024 GiB Maxscore is the SBS. For 2048 GiB and 4096 GiB it is VBMW.

Absolute VBS and SBS latency are shown in Figure 4.3. To visualize the size of the gap itself more clearly, we show the absolute latency gap in Figure 4.5. The absolute gap shows how much time that potentially, on average, can be saved per query. The latency overhead of an algorithm selector needs to be lower than the absolute gap for a speedup to be possible. The relative VBS-SBS gap is shown in Figure 4.6. The relative gap is directly translatable to the largest potential speedup using the algorithm selection technique.

Although Figure 4.5 shows potential latency improvements for all collection sizes, the absolute latency gap is small for smaller collection sizes. However, for the larger collection sizes, there is some potential for improvement. The trend of an increasing VBS-SBS gap when scaling up the collection size is promising as large dataset sizes are relevant for real-world applications. Based on the relative gap observed in Figure 4.6 the largest potential speedup observed is for the collection size 2048 GiB, with a VBS-SBS gap of 24.6%. This indicates clear complementarity between algorithms, as an answer to RQ1. In the rest of the Results section the index with collection size 2048 GiB is used for deeper analyses. This collection size was selected, as the algorithm selection technique has the most potential here and therefore leads to more interesting results in analyses.

An interesting detail of this data is that the absolute gap increases from 2048 GiB to 4096 GiB, while the relative gap decreases quite significantly. The reason for this is that 2048 GiB is near a peak in relative VBS-SBS gap because here the different algorithms are the most even, in terms of how many queries they are the best in. Therefore, the VBS can be significantly better than the SBS, by combining the different algorithms. 2048 GiB is the first collection size where VBMW is the SBS instead of Maxscore and this trend continues the larger the collection size gets. Thus, at 4096 GiB VBMW is more dominating, and therefore the relative gap gets

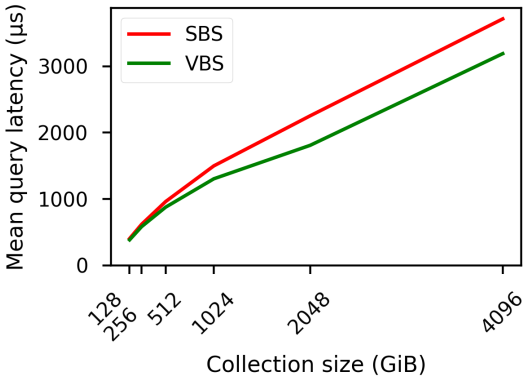

Figure 4.3: Average VBS and SBS latency - The VBS-SBS gap, for different collection sizes

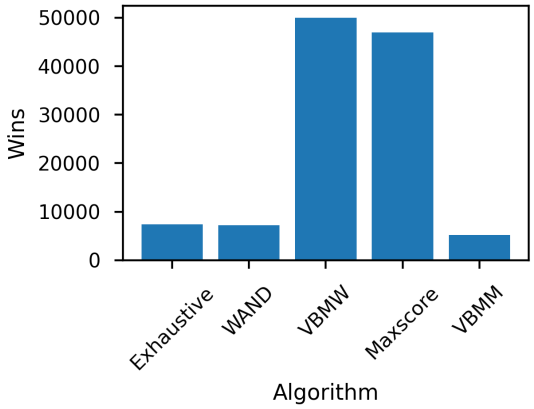

Figure 4.4: Distribution of wins over algorithms (2048 GiB)

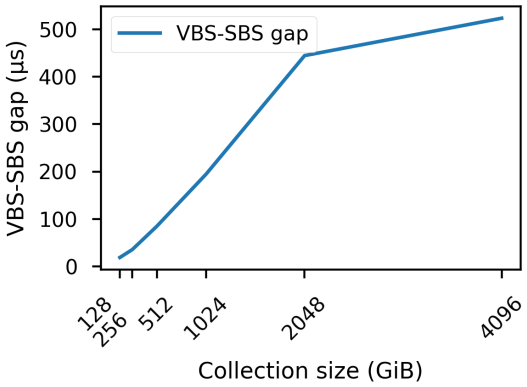

Figure 4.5: Absolute VBS-SBS gap

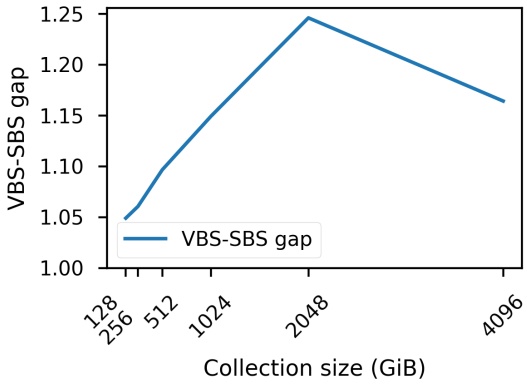

Figure 4.6: Relative VBS-SBS gap, i.e. potential speedup

smaller again. The absolute gap is still larger though. This is because the latencies are overall greater for larger collection sizes because it takes longer to search a large index.

4.3.1 Algorithms Contribution to the Gap

Here we present to which degree the algorithms contribute to the VBS-SBS gap. This refers to how many of the queries the VBS used each available algorithm, i.e. the number of queries where each algorithm was the fastest. We also refer to this as the number of wins by each algorithm. In Figure 4.4, the distribution of wins over the algorithms is presented for the collection size 2048 GiB. Here VBMW has the most wins, which naturally coincides with the fact that it is the SBS. A large part of the wins are split between VBMW and Maxscore, which is the SBS for smaller collection sizes. For smaller sizes, the distribution would be slightly shifted towards Maxscore, with the most wins for Maxscore as it is the SBS. The more primitive algorithms Exhaustive DaaT and WAND also have some contribution to the gap. VBMM has a small contribution as well. It should be noted that the wins, in Figure 4.4, are

not adjusted for the significance of how much latency each win would contribute to decrease. Therefore it should be seen as an indication of what algorithms play a role in reducing the query latency. In section 4.5.3 the weight of individual wins are considered.

4.4 Feature Quality Analysis

Here we present the results of the feature quality analysis described in section 3.3.2. This is done to analyze to what degree the features can provide sufficient information for a machine learning model to map search instances to the best algorithms. This analysis also indicates the answer to RQ2 - ML model mapping. The features are k, term count, the range of PLL and TMS values, as well as the summary features average, median, sum, maximum, and minimum for the PLLs and TMSs.

4.4.1 Feature Informativeness

First, we present the results of the informativeness analysis. This aimed to analyze whether the features encapsulate enough information about the queries to enable mapping to which the best algorithm would be. To analyze how informative the features are, we fitted a random forest model, without any complexity restrictions, on the data and studied the accuracy it was able to achieve. The resulting accuracy was 80.6% for the 2048 GiB collection size. This means that while the features are informative, and enable the model to map 80.6% of the instances to the optimal algorithm, there are still 19.4% of the instances where the features are not informative enough for the model to be able to accurately classify which algorithm performs the best.

As a second step, we calculated the feature importance of the features, to see which of the individual features are the most informative. The feature importance for 2048 GiB is shown in Figure 4.7. As the figure shows, the term count is the most important feature. This is explained by the fact that VBMW is the most frequently best algorithm for 1- and 2-term queries, while Maxscore is the best in general for queries of more than 2 terms. These are the two most important algorithms and therefore the term count provides an informative split between the two. Confirmation of this is shown in Figure 4.8. Subsequently, we have the maximum summary feature of both the PLLs and the TMSs. These features enable more distinction within the split of the term count. The importance of these features though, is less than half of the importance of the term count. After these, we see the rest of the summary features, and then k. k is the last feature we consider to be useful. This limit to which features we consider useful is marked in Figure 4.7 with a red line. Thus, we do not consider minimum and median to be productive summary features. We also do not consider the individual PLL and TMS values to be useful features.

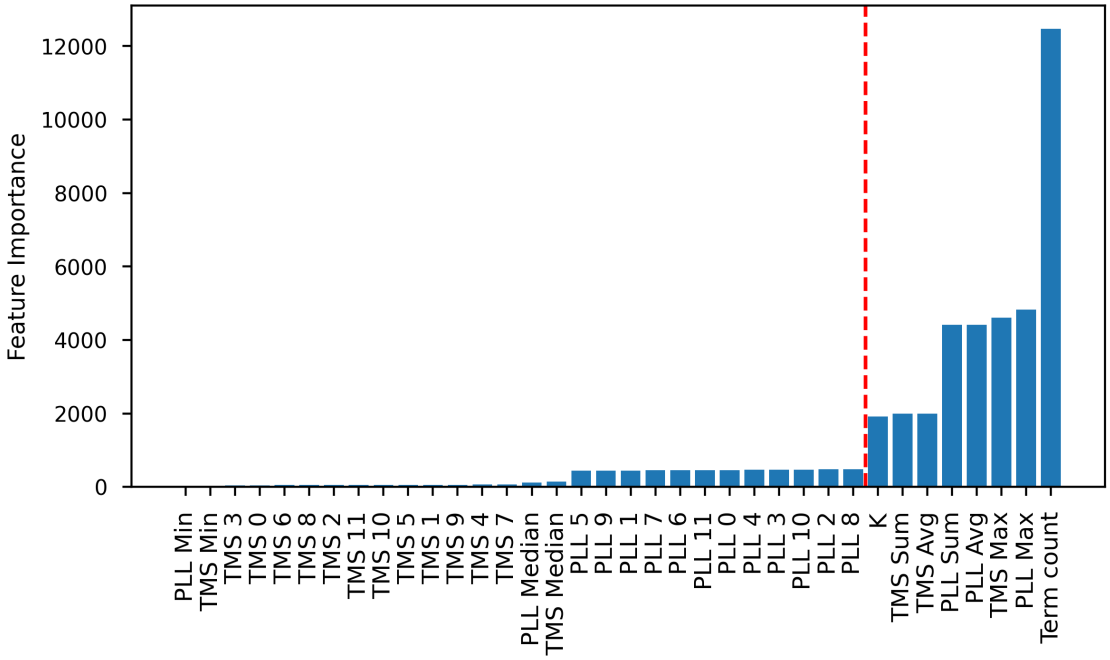

Figure 4.7: F-statistic based feature importance (2048 GiB). The red line denotes the limit to which features we consider useful.

This analysis showed that the individual PLLs and TMSs, and their median and minimum were not sufficiently informative as features. These features were not included in further analyses, to focus more on the most informative features.

4.4.2 Feature Complementarity

To analyze the complementarity of the features we looked at the Pearson correlation matrix. The correlation matrix of the features for the 2048 GiB collection is displayed in Figure 4.9. It shows to what degree a pair of features correlate linearly. 1 means full positive linear correlation, -1 means full negative linear correlation, and 0 means no correlation. The matrix shows that the summary features mean, sum and maximum are all highly correlated. The maximum summary feature is slightly less correlated to the other features than the other summary features, especially for the TMSs. The positive correlation between the summary features is logical, as e.g. a larger sum value would lead to a larger mean value and generally a larger max.

When it comes to the term count it has a weaker but still a slight correlation to the summary features. The correlation between the sum and the maximum is expected as both will become larger as the number of terms increases. k does not correlate with any of the other features. This could make it a useful feature, even if it did not have a high feature importance value, because it provides unique information that no other feature encapsulates.

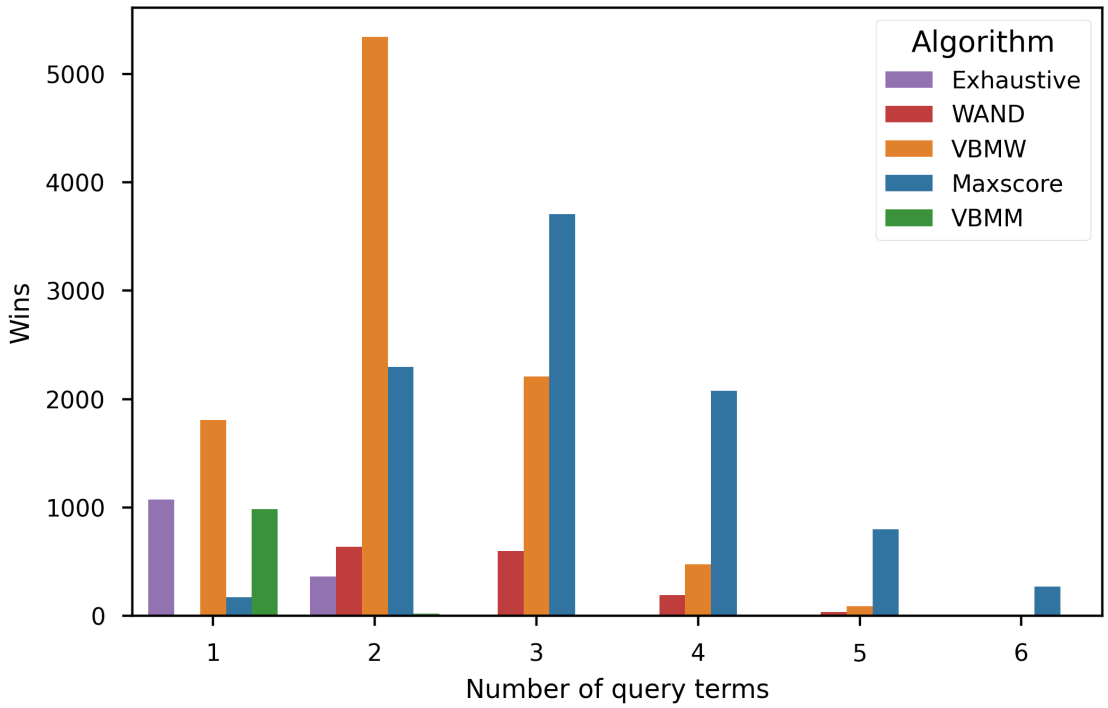

Figure 4.8: Number of wins per algorithm distributed over term counts (2048 GiB)

4.4.3 Feature Computation Cost

To analyze how cheaply computable the features are, we measured the computation time for the following feature sets: only the PLL-based features, only the TMS-based features, and all features together (see section 3.3.2). Because the computation time complexity is linear to the number of query terms, we analyzed the results per term count. The results are shown in Figure 4.10.

First, we note that the TMS-based features had a measured computation time of 0 microseconds (μs) in all instances. This is likely due to the presence of this information in the cache at access time. This causes the load times, which is the most time-consuming part of the computation of these features, to be below the μs resolution of the measurements. The difference in TMS computation time to the PLLs is likely an implementation-specific behavior, as the PISA implementation stores the PLLs separately from the TMSs. This might differ in other implementations, as both the TMSs and PLLs can be considered to be part of the vocabulary of the inverted index.

For the first and third feature sets (PLL-based features and all features), defined in section 3.3.2, feature computation times were also very short. Feature computation for queries with less than 8 terms took less than 4 μs. The features for queries with 10 or more terms took longer, but still under 10 μs to compute. With feature computation times this short, we consider them negligible and did not account for them during algorithm selector prototyping.

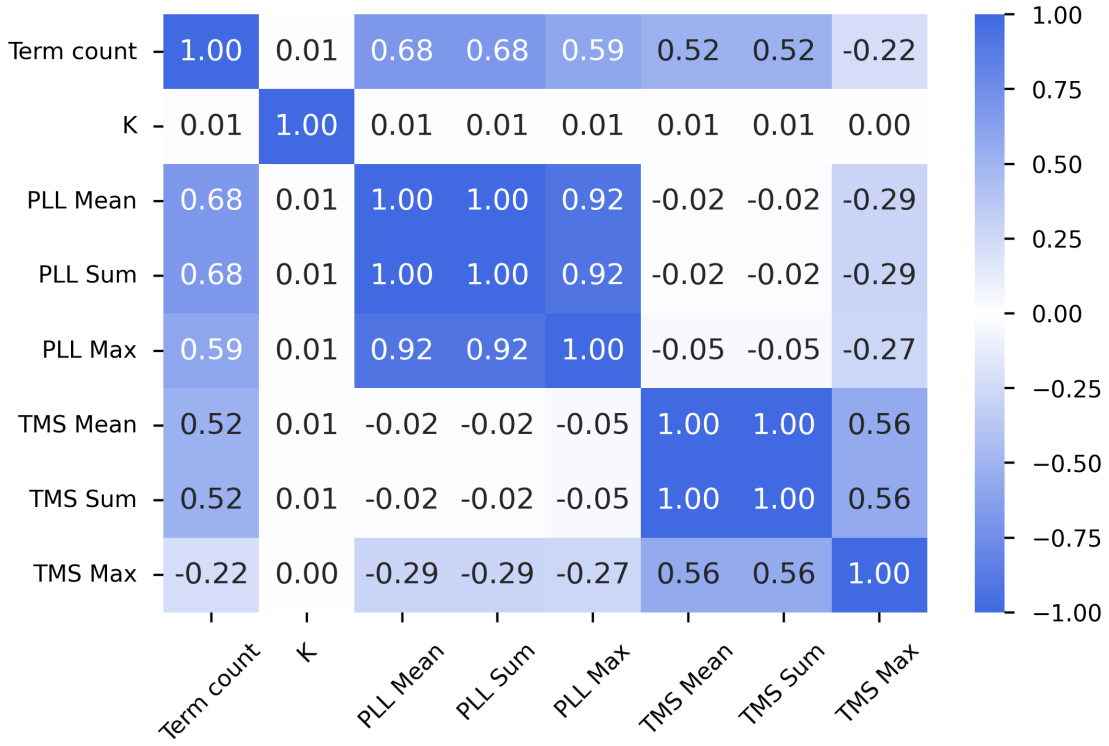

Figure 4.9: Feature complementarity analysis - Feature correlation matrix

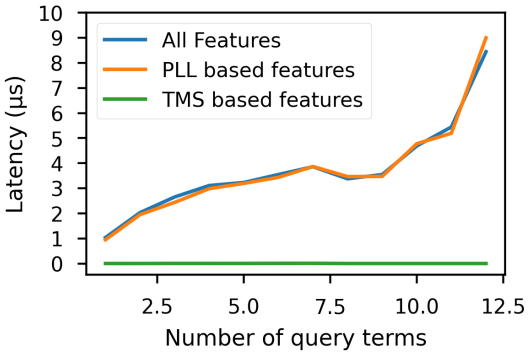

Figure 4.10: Feature computation times for only PLL-based features, only TMS-based features, and all features

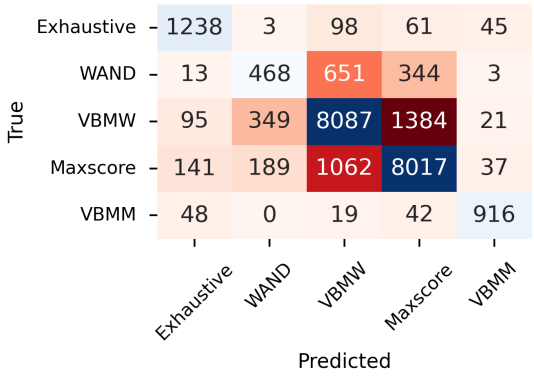

Figure 4.11: Feature applicability analysis - Confusion matrix (2048 GiB)

4.4.4 Feature Applicability

When it comes to how generally applicable the features are, we refer back to the accuracy of the random forest model discussed in the feature informativeness analysis in section 4.4.1. The accuracy was 80.6% and therefore the features are not applicable to all instances. To analyze further whether any specific type of query stands for this gap in accuracy, we produce a confusion matrix over the predictions of the model on a test dataset. The confusion matrix (see Figure 4.11) shows the

number of instances predicted to be each algorithm, for each different value of true algorithm. The correctly classified instances lie on the diagonal where the predicted and the true algorithm are the same. Here we see that the majority of misclassified instances are between Maxscore and VBMW. The mean value of the term count in these misclassified instances was 2.51. This indicates, as was observed in the informativeness analysis above (section 4.4.1), that the term count feature does not provide a completely clean split between Maxscore and VBMW, and the other features are not able to entirely map instances where the term count is around 2 and 3. We also note that there are cases of miss-predictions between VBMW and WAND. This is likely because they have similar performance characteristics because VBMW is based on WAND. To conclude, we claim that the features are not entirely generally applicable, and there are specific types of queries that are more likely to be misclassified.

4.4.5 Feature Quality Analysis Summary

To summarize the feature quality analysis, we consider the available features to be cheap to compute, but not fully informative. In some cases, they do not provide enough information for a classifier to judge which algorithm will be the fastest. Particularly in the case when the number of query terms is 2 or 3. Because of this, a classifier cannot just focus on maximizing the selection accuracy, because this cannot reach near 100%. Since the highest achievable accuracy is around 80% with these features, there are many different ways to map 80% of the instances. And in the end accuracy in itself does not necessarily matter. Consequently, the classifier must target the resulting latency of the algorithm selections, rather than the classification accuracy. Misclassifying an instance might not be significant if the chosen algorithm was almost as good as the optimal. But on the other hand, choosing the wrong algorithm in some cases could be very costly. The classifier must therefore be trained to avoid costly misses and not be penalized too much for instances where the second-best algorithm is sufficient. This will be elaborated on in the following section 4.5.3 on instance weighted classification.

There is a large correlation within the summary features, particularly between mean and sum, which indicates that they encode the same or similar information. Therefore it may be redundant or even adverse to compute all summary features. k on the other hand, was not correlated to any of the other features. With this in mind, a reasonable feature set could be: term count, k, maximum PLL, and maximum TMS. This is the set of features that we use to train our classifiers. This analysis also indicated that a simple ML model has good potential to map the relation between search instances and best algorithms, providing a first answer to RQ2 - ML model mapping.

4.5 Classifiers

In this section, we present the results of the development of the classification model that lies at the heart of the algorithm selector. First, we present the results of the

latency overhead of the two implementation candidates in section 4.5.1. Then we present the results of hyper-parameter optimization of the classifier model in section 4.5.2. The four features that are used to train the classifiers are, as described in section 4.4.5: term count, k, maximum PLL, and maximum TMS. In section 4.5.3, we describe the instance weight technique, which is crucial to achieve good performance in the algorithm selector. In section 4.5.4 we present the resulting decision tree classifier. Finally, in section 4.5.5 we summarise the classifier development results.

4.5.1 Implementation Latency

First, we evaluated the viability of the Library implementation (section 3.3.3) of random forest, to see if this achieves a low enough overhead. To test this, we grid searched over the parameters in table 3.2. The maximum depth is limited to 8. This limitation was because of unreasonable code size and compile times of deeper trees. The evaluation was performed on the largest collection size, 4096 GiB. We used this size because the absolute VBS-SBS gap is largest here (approximately 500 μs), which means that there is the most tolerance for selection overhead. The best performing hyper-parameter configuration was: tree depth: 8, and number of estimators: 4.

Using the predictions of this model we calculated what algorithm selections it would make on a test set of benchmark results. We then calculated what speedup it would result in, by summing the latencies of the selected algorithms, measured during benchmarking, and adding the latency overhead for each instance. The resulting calculated expected speedup compared to the SBS VBMW, was 0.93. This means that the best algorithm selector actually would perform worse than the SBS, even though it achieved a high accuracy of 80.2%. This was because the latency of the machine learning model was measured to be 652 μs. As this is larger than the whole VBS-SBS gap, a selector using this classifier would not be able to close any fraction of the gap. Thus, we conclude that the library implementation of random forest is too slow for this application.

We then evaluated the latency overhead of the simpler If-statement tree implementation (section 3.3.3). We ran benchmarks with decision trees of the depths presented in table 3.3 and measured the classification latency. The classification latency was measured with a μs resolution. The results showed that trees of depths under 10 did not have a single occasion where the classification took even a single μs. For the deeper trees, there were instances where classification took up to 15 μs, but most executions were measured to 0 μs. The average classification latency for the deepest tree, with a maximum depth of 20, was 0.02 μs. As a result of the very low latency overhead, this implementation was chosen.

4.5.2 Hyper-parameter Optimization

Because of the low latency overhead of the If-statement implementation, we did not add any latency overhead when grid searching for the best hyper-parameter configuration. Due to a negligible latency overhead, the trade-off between this and algorithm latency was irrelevant for the If-statement implementation. The configuration of hyper-parameters is therefore a direct result of optimization for algorithm latency solely.

Grid searches were performed for each collection size and with the grid specified in table 3.3. The parameter for the number of estimators does not apply here since this implementation supports only decision trees and not random forests. The grid search results showed that, in general, a depth of 12 performed the best. Differences in hyper-parameter configuration probably come from small differences in the characteristics of the data, but in general, similar hyper-parameter configurations performed the best for all collection sizes. This implementation with a maximum depth of 12 was used in the final evaluation of the decision tree algorithm selectors in section 4.6.1.

4.5.3 Instance Weighted Classification

By default during training, the decision trees do not consider that selecting the best algorithm can be of varying importance depending on the possible performance gain of each selection. For example, for some queries, all algorithms have similar query times. In these cases, it is not very important to select the very best algorithm. But for other queries, the best algorithm can be significantly faster than the others. Our evaluations showed that without considering this, the algorithm selectors did not perform well, sometimes worse than the SBS. Because the features are not informative enough to correctly map all queries, a successful approach needs to account for this.

Because classification without considering the varying significance of the different instances did not perform well, we introduced the concept of weighted instances in the training. Each instance, or query in this case, in the training data, was assigned a weight signifying the importance of choosing the best algorithm in that query. The weights are calculated as the difference in latency between the best-performing t_0 and second-best-performing t_1 algorithms, as defined below. This weight therefore measures how much latency we can save by selecting the best algorithm.

$$w = t_1 - t_0$$

When training the decision trees with these instance weights, they performed better. But speedups were still very small, reaching a speedup of around 3% for the 2048 GiB collection where a 25% speedup is theoretically achievable. To simplify the decisions of the model and remove the risk of choosing algorithms that are very slow in some cases we evaluated a classifier that chooses only between VBMW and Maxscore. As can be seen in Figure 4.4 these algorithms contribute to the vast majority of all wins and they also rarely perform very badly. Therefore this change simplifies the model and reduces the risk of undesirable selections, at a low cost. With this simplification,

the models were able to perform significantly better. The results will be elaborated on in the evaluation in section 4.6.

4.5.4 Decision Tree Interpretation

To interpret the trained decision tree, we plot the first 3 levels of the decision tree, trained on data from the index of 2048 GiB collection size. Only the first three levels of the tree are shown as it grows very large for deeper levels. The plot is shown in Figure 4.12. The figure shows each node in the tree, which feature it splits the input on, and the threshold value. The upper children are selected for inputs greater or equal to the given threshold and the lower children are selected for inputs smaller than the threshold. From the tree plot, we see that the decision at the root of the tree uses the maximum PLL feature. This is somewhat unexpected as the term count seems to be the feature resulting in the best split. This feature is however used slightly lower in the tree, on the third level.

To summarize the leaves or output of the tree, we looked at the accuracy of the model on test data. This reached 78.7%. As we have discussed though, this is not an entirely useful metric, as it is significant which of the instances are correctly classified. To get an idea of this we calculated the mean instance weight of the correctly classified instances and the mean weight of the misclassified instances. The mean weight of the correctly classified instances was 1063 and that of the misclassified was 297. This shows that the tree was able to prioritize significantly more important instances.

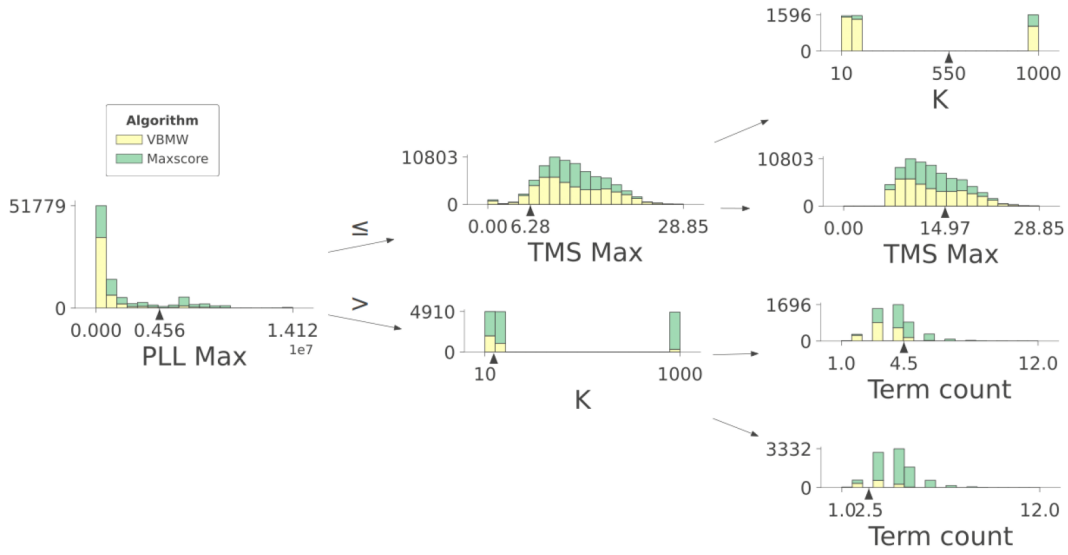

Figure 4.12: Decision tree plot, levels 1 - 3 (2048 GiB)

4.5.5 Classifiers Summary

To conclude, we have shown that the TL2cgen library implementation of random forest and decision tree did not have a low enough latency to be viable for algorithm

selection in this scenario. As decision trees are very lightweight machine learning models, and the TL2cgen C++ implementation is considered efficient, it is likely that most "off-the-shelf" implementations are too slow. Therefore we designed a very low latency implementation of decision tree using nested if-statements. This implementation was able to achieve a sub-μs latency in the majority of executions. This is ideal as practically none of the VBS-SBS gap is then wasted on classification overhead.

We also performed a hyper-parameter optimization of this decision tree. A maximum depth of 12 performed the best. This is good as it is not too deep. This leads to manageable C++ code sizes and models that can be interpreted to a reasonable degree.

Finally, we also determined that the individual importance of the query instances needs to be taken into account when training decision trees. Otherwise, the model will be indifferent to how much potential latency improvement different kinds of queries have, and will not perform well. We therefore calculated instance weights for the training samples, which led to models prioritizing the right kinds of queries to classify correctly. This leads to far better performance. The final results of the performance of the algorithm selectors will be evaluated in section 4.6 below.

4.6 Evaluation

Here the final performance results of the C++ implementations of the models integrated in the experimental setup as described in section 4.1 are presented. In section 4.6.1 the performance of the developed algorithm selectors is presented for each collection size. The algorithm selector implementations are evaluated on the same inverted indexes as it was trained on. Finally, in section 4.6.2 we present the performance of the algorithm selectors on inverted indexes that they were not trained on data from, to see how well a trained model generalizes to change in the inverted index.

4.6.1 Decision Tree Algorithm Selector Evaluation

Here we present the performance results of the developed decision tree algorithm selectors. This is the simpler tree of if-statements implementation, introduced in section 3.3.4, analyzed in section 4.5.1, and the hyper-parameter configurations are presented in section 4.5.2. The selectors are compared against the state-of-the-art query processing algorithms Maxscore and VBMW. These are, according to our results, the two best-performing existing algorithms. Performance is shown for every collection size. The results of each collection size show the algorithm selector performance when trained on benchmark data from that collection size.

The results in Figure 4.13 show that the algorithm selectors were able to outperform the SBS for every collection size. The largest speedup was, as expected, at 2048 GiB where the algorithm selector achieved a speedup of 17.7%. The speedups were, in the order of collection size: 2.0%, 4.0%, 6.7%, 12.1%, 17.7%, and 12.1%. This

amounts to closing 40.8%, 66.7%, 69.1%, 81.2%, 72.0%, and 73.8% of the VBS-SBS gaps. These results provide a full answer to, RQ2 - ML model mapping, showing that a simpler ML model can map the relationship between search instances and the best algorithm, to a large degree. The speedups also appear significant enough to justify the approach, even though they are very small for indexes with small collection sizes. In Figure 4.14 we visualize how the algorithm selectors close the fractions of the VBS-SBS gaps. The colored, blue area marks the fraction of the VBS-SBS gaps that the selectors manage to close.

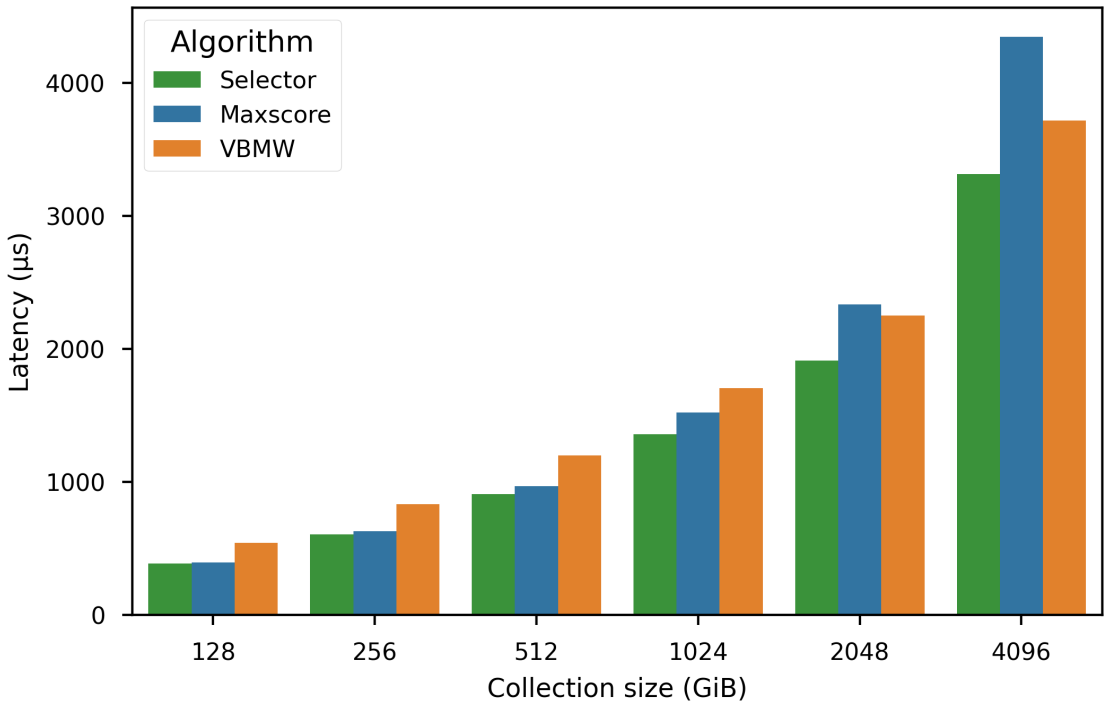

Figure 4.13: Decision tree algorithm selection latency per collection size. Comparison with Maxscore and VBMW query latency.

4.6.2 Robustness to Change in Index Size

In Figure 4.15 we show how each of the algorithm selectors performs on different index sizes than the original sizes they were trained on. This is done to evaluate how the trained classifiers can handle change in the inverted index, such as when the database is expanded or shrunk. This provides an answer to RQ3 - Robustness to change in index size.

Every algorithm selector was evaluated on every collection size. The performance of every selector is plotted as a line of their speedup compared to the SBS at every collection size. The SBS is plotted as a red line at 1, for reference. The circles indicate points where the selector was evaluated on the same inverted index as it

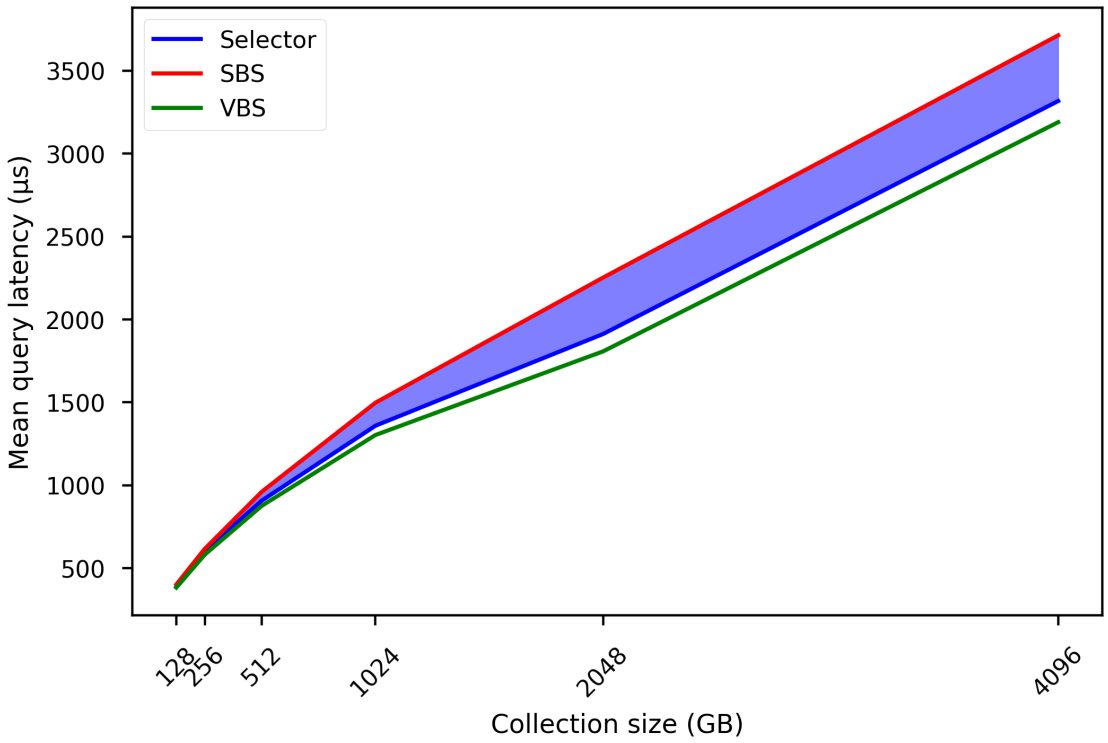

Figure 4.14: VBS-SBS gaps with algorithm selector latency per collection size. The fraction of the gap closed is marked in blue.

was trained on.

As can be observed from Figure 4.15 at the circled points, each selector did, as expected, perform the best at the collection size they were trained on. In general, the lines slope downwards from the points they were trained on, meaning that they did not perform as well on indexes of collection sizes that they were not trained for. Some exceptions include for example the 4096 GiB selector having a larger speedup on 2048 GiB. This is however simply due to the fact that the VBS-SBS gap is larger at 2048 GiB than 4096. It should be noted that the values on the collection size axis increase exponentially.

Finally, as an answer to RQ3 - Robustness to change in index size, we conclude that the algorithm selectors perform the best on the collection size they were trained on. Therefore, in a realistic environment where the index most likely changes in size over time, this needs to be taken into account. Small changes appear to be tolerable, but large changes in collection size will significantly affect performance.

4.7 Energy Consumption Estimation

When estimating energy consumption for the processes performing top-k document retrieval, we assume constant energy consumption of the system during utilization

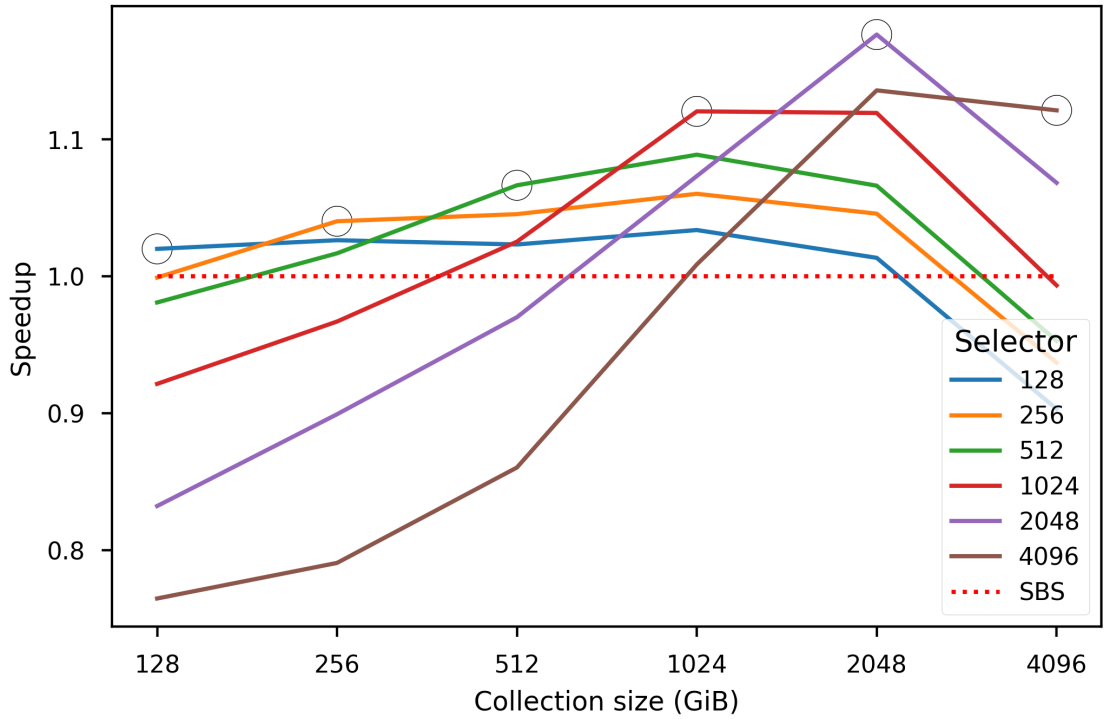

Figure 4.15: Robustness to change in index collection size. The circles indicate points where the selector was evaluated on the same inverted index as it was trained on.

time. By this assumption, the system consumes power P when utilized. This follows the linear relation between energy consumption and query latency, shown in [2]. E_{SBS} and E_{AS} refer to the average energy consumption for processing 1 query with the SBS and with an algorithm selector. t_{SBS} and t_{AS} refer to the average execution time for processing 1 query with each approach respectively. The speedup of the algorithm selector over the SBS is:

$$S_{AS} = \frac{t_{SBS}}{t_{AS}}$$

The reduction in energy consumption can then be defined as below:

$$\frac{E_{SBS} - E_{AS}}{E_{SBS}} = 1 - \frac{P \cdot t_{AS}}{P \cdot t_{SBS}} = 1 - \frac{1}{S_{AS}}$$

With the above assumptions, the reduction in energy consumption is directly dependent on the speedup. This results in a reduction in energy consumption of 15.0% for the best-performing algorithm selector at the 2048 GiB inverted index, with a speedup of 17.7%. This means that, according to this approximation, by using the suggested algorithm selection approach instead of the SBS, VBMW, on the 2048 GiB index, the query process would consume 15.0% less energy.

4.8 Discussion

In this chapter, we have presented the results of the thesis. We have seen that algorithm selection for top-k document retrieval is an area where the margins for new speedup are very thin. We have shown that the latency of standard ML library implementations is too slow for this application and more basic and optimized implementations are necessary. Furthermore, the features that we have evaluated are not fully informative and generally applicable to achieve satisfactory classification accuracy. This complicates things further as the range of queries that can be correctly predicted has to be chosen carefully. Still, we have shown that some latency improvements are achievable, particularly within the larger collection sizes tested. We were able to achieve speedups of up to 17.7%. According to our energy consumption estimate, this could reduce the energy consumption of the top-k document retrieval process by 15.0%. Although the algorithm selection approach was not able to achieve significant speedups at the lower collection sizes, it did at least not perform worse than the SBS. This is advantageous as less care has to be taken about when to use the approach, than if it could be adverse in some cases.

Generally, the speedup depends entirely on the VBS-SBS gap. For most of the collection sizes the algorithm selectors were able to close around 70% of the gap, and the speedup then follows from the size of the gap. As Figure 4.6 shows though, likely, the peak of the relative VBS-SBS gap of the top-k document retrieval problem lies somewhere between 1024 and 4096 GiB collection sizes. It is therefore unlikely that potential speedups of much more than 25% can be found.

When it comes to the practical aspects of the algorithm selection approach, we have shown in Figure 4.15, that an algorithm selector is more or less specific to the collection size that it was trained on. Because of this, if algorithm selection were to be used in a realistic scenario, some strategy would have to be adopted for how to manage the potential re-trainings that could be necessary, if the inverted index is updated beyond the range that the selector can generalize to. Figure 4.15 shows though, that the selectors can handle changes in collection size of at least up to a TiB fairly well. Re-training the model is also not particularly expensive. A sufficient number of queries, with varying characteristics, have to be run and timed to get enough instances to train the machine learning model on, without over-fitting it to those particular instances. If this needs to be done often, however, this overhead could negate the energy consumption gains of the approach. An alternative approach could be to measure the latency of queries intermittently, and this data could occasionally be used to re-train the model during off-hours.

If the complexity of an ML-based algorithm selector is not desirable, a simple version of this could still be implemented without having to train a model on data. As Figure 4.8 shows, VBMW is dominant for 1- and 2-term queries, and Maxscore is dominant for queries with three or more terms. Therefore, a very simple yet profitable version of this approach could be to simply use an if-statement on the term count to select between VBMW and Maxscore, at the 2.5 query terms threshold.

This relationship is also independent of collection size. Therefore, this approach would allow for an easier implementation of algorithm selection, without the need to train an ML model, at the cost of worse performance.

5

Conclusions

In this thesis, we studied whether algorithm selection is a viable approach for improving query latency in top-k document retrieval, for web crawl data. This was done by first benchmarking existing query processing algorithms to record their performance. This data was then used to show the potential of the algorithm selection approach, as well as ML training data for algorithm selectors. Finally, the algorithm selectors were implemented in C++ and evaluated against state-of-the-art query processing algorithms. The algorithm selector approach was able to surpass the performance of the state-of-the-art algorithms, with query latency speedups of up to 17.7%, when used on an index with 2048 GiB of web crawl data. According to our approximations, this could translate to a reduction in energy consumption of 15.0%. It shows that algorithm selection can be a beneficial approach for lowering energy consumption in top-k document retrieval.

5.1 Research Questions

The goal of showing the viability of the algorithm selection approach was divided into 3 research questions. To answer RQ1 - Complementarity between algorithms, the performance results of the query processing algorithms were analyzed using the VBS-SBS gap metric. This showed a potential speedup of 24.6% for the index of size 2048 GiB if an algorithm selector would select the optimal algorithm every time.

To answer RQ2 - ML model mapping, we trained and tested random forest models of varying complexity. The best performing model achieved an accuracy of up to 80.6% for the 2048 GiB collection size. This shows that the algorithms can be mapped to the search instances using the given features, although not perfectly.

To see how the approach generalizes to other inverted indexes than the one used for the training data, RQ3 - Robustness to change in index size, the algorithms were evaluated on the other inverted indexes that they were not trained for. While the algorithm selectors were able to handle some change, this showed performance to be dependent on index size. With this, we conclude that the classifier model of an algorithm selector has to be retrained to adapt to the new index if it significantly expands or shrinks in size.

In conclusion, we show a potential speedup for RQ1, achieve high accuracy for RQ2, and lastly show the need for retraining for different index sizes for RQ3. In total this shows algorithm selection to be a viable approach for top-k document retrieval on web crawl data.

5.2 Future Work

Future work includes both expanding the domain where the algorithm selection approach is applied and evaluated, as well as fine-tuning and investigating new techniques for creating the algorithm selector itself. We have evaluated the approach by using a fixed index setup, a fixed data set, and query data set. The domain of usage could be widened by trying other index configurations and using different sets of textual data.

When it comes to techniques for training the algorithm selector, improvements could possibly be made on what features to select to provide the algorithm selector with more information to base its decision on. To further investigate more advanced, but at the same time well optimized, implementation techniques could be beneficial. This would enable a more complex ML algorithm and possibly could lead to a greater speedup.

Finally, an algorithm selection technique, where the execution time of multiple algorithms is predicted instead of the algorithm itself, could be evaluated. Here, the predicted execution times are used to decide which algorithm potentially performs best. This way, 1 complex classification problem is reduced to n possibly simpler regression problems.

5.3 Final Conclusions

The final conclusions of this thesis is that algorithm selection is a viable approach for top-k document retrieval on web crawl data when implemented as a decision tree in the if-statement format. The resulting speedup of 17.7% potentially translates to a 15.0% reduction in energy consumption spent on query processing.

If this approach were to be applied to search engine systems on a large scale it would lead to a great total reduction in absolute energy consumption. As energy consumption in data centers continuously increases, this absolute reduction would become larger and more relevant over time.

Bibliography

[1] A. S. G. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, pp. 117–157, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:110716199.

[2] C. M. and T. N., "A study on query energy consumption in web search engines," in *6th Italian Information Retrieval Workshop, Cagliari, Italy, 25-26/05/2015*, M. Jeusfeld c/o Redaktion Sun SITE, Informatik V, RWTH Aachen., Aachen, Germania, 2015.

[3] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, ser. CIKM '03, New Orleans, LA, USA: Association for Computing Machinery, 2003, pp. 426–434, ISBN: 1581137230. DOI: 10.1145/956863.956944. [Online]. Available: https://doi.org/10.1145/956863.956944.

[4] H. Turtle and J. Flood, "Query evaluation: Strategies and optimizations," *Information Processing & Management*, vol. 31, no. 6, pp. 831–850, 1995, ISSN: 0306-4573. DOI: https://doi.org/10.1016/0306-4573(95)00020-H. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030645739500020H.

[5] A. Mallia, M. Siedlaczek, and T. Suel, "An experimental study of index compression and daat query processing methods," in *Advances in Information Retrieval: 41st European Conference on IR Research, ECIR 2019, Cologne, Germany, April 14–18, 2019, Proceedings, Part I*, Cologne, Germany: Springer-Verlag, 2019, pp. 353–368, ISBN: 978-3-030-15711-1. DOI: 10.1007/978-3-030-15712-8_23. [Online]. Available: https://doi.org/10.1007/978-3-030-15712-8_23.

[6] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman, "A comparison of document-at-a-time and score-at-a-time query evaluation," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM '17, Cambridge, United Kingdom: Association for Computing Machinery, 2017, pp. 201–210, ISBN: 9781450346757. DOI: 10.1145/3018661.3018726. [Online]. Available: https://doi.org/10.1145/3018661.3018726.

[7] B. Croft, D. Metzler, and T. Strohman, *Search Engines: Information Retrieval in Practice*, 1st. USA: Addison-Wesley Publishing Company, 2009, ISBN: 0136072240.

[8] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, 6–es, Jul. 2006, ISSN: 0360-0300. DOI: 10.1145/1132956.1132959. [Online]. Available: https://doi.org/10.1145/1132956.1132959.

[9] V. N. Anh and A. Moffat, "Index compression using fixed binary codewords," in *Fifteenth Australasian Database Conference (ADC2004)*, K.-D. Schewe and H. E. Williams, Eds., ser. CRPIT, vol. 27, Dunedin, New Zealand: ACS, 2004, pp. 61–67.

[10] H. E. Williams and J. Zobel, "Compressing Integers for Fast File Access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, Jan. 1999, ISSN: 0010-4620. DOI: 10.1093/comjnl/42.3.193. eprint: https://academic.oup.com/comjnl/article-pdf/42/3/193/962527/420193.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/42.3.193.

[11] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi, "Simd-based decoding of posting lists," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11, Glasgow, Scotland, UK: Association for Computing Machinery, 2011, pp. 317–326, ISBN: 9781450307178. DOI: 10.1145/2063576.2063627. [Online]. Available: https://doi.org/10.1145/2063576.2063627.

[12] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012, ch. 1.3.3 Vector and array processors, p. 458.

[13] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw. Pract. Exper.*, vol. 45, no. 1, pp. 1–29, Jan. 2015, ISSN: 0038-0644. DOI: 10.1002/spe.2203. [Online]. Available: https://doi.org/10.1002/spe.2203.

[14] A. Trotman, "Compression, simd, and postings lists," in *Proceedings of the 19th Australasian Document Computing Symposium*, ser. ADCS '14, Melbourne, VIC, Australia: Association for Computing Machinery, 2014, pp. 50–57, ISBN: 9781450330008. DOI: 10.1145/2682862.2682870. [Online]. Available: https://doi.org/10.1145/2682862.2682870.

[15] D. K. Blandford and G. E. Blelloch, "Index compression through document reordering," *Proceedings DCC 2002. Data Compression Conference*, pp. 342–351, 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:8092524.

[16] F. Silvestri, "Sorting out the document identifier assignment problem," in *Proceedings of the 29th European Conference on IR Research*, ser. ECIR'07, Rome, Italy: Springer-Verlag, 2007, pp. 101–112, ISBN: 9783540714941.

[17] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, "Compressing graphs and indexes with recursive graph bisection," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, ACM, Aug. 2016. DOI: 10.1145/2939672.2939862. [Online]. Available: http://dx.doi.org/10.1145/2939672.2939862.

[18] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes," in

The 37th International Conference on Very Large Databases (VLDB 2011), 2011.

[19] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at trec-3.," Jan. 1994.

[20] S. Ding and T. Suel, "Faster top-k document retrieval using block-max indexes," in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '11, Beijing, China: Association for Computing Machinery, 2011, pp. 993–1002, ISBN: 9781450307574. DOI: 10.1145/2009916.2010048. [Online]. Available: https://doi.org/10.1145/2009916.2010048.

[21] N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient query processing for scalable web search," *Foundations and Trends® in Information Retrieval*, vol. 12, no. 4-5, pp. 319–500, 2018, ISSN: 1554-0669. DOI: 10.1561/1500000057. [Online]. Available: http://dx.doi.org/10.1561/1500000057.

[22] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini, "Faster blockmax wand with variable-sized blocks," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '17, Shinjuku, Tokyo, Japan: Association for Computing Machinery, 2017, pp. 625–634, ISBN: 9781450350228. DOI: 10.1145/3077136.3080780. [Online]. Available: https://doi.org/10.1145/3077136.3080780.

[23] D. Shan, S. Ding, J. He, H. Yan, and X. Li, "Optimized top-k processing with global page scores on block-max indexes," in *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, ser. WSDM '12, Seattle, Washington, USA: Association for Computing Machinery, 2012, pp. 423–432, ISBN: 9781450307475. DOI: 10.1145/2124295.2124346. [Online]. Available: https://doi.org/10.1145/2124295.2124346.

[24] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated Algorithm Selection: Survey and Perspectives," *Evolutionary Computation*, vol. 27, no. 1, pp. 3–45, Mar. 2019, ISSN: 1063-6560. DOI: 10.1162/evco_a_00242. eprint: https://direct.mit.edu/evco/article-pdf/27/1/3/1552398/evco_a_00242.pdf. [Online]. Available: https://doi.org/10.1162/evco%5C_a%5C_00242.

[25] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press, 2022. [Online]. Available: https://smlbook.org.

[26] L. Kotthoff, "LLAMA: leveraging learning to automatically manage algorithms," *CoRR*, vol. abs/1306.1031, 2013. arXiv: 1306.1031. [Online]. Available: http://arxiv.org/abs/1306.1031.

[27] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, 2018, ISSN: 0003-2999. [Online]. Available: https://journals.lww.com/anesthesia-analgesia/fulltext/2018/05000/correlation_coefficients__appropriate_use_and.50.aspx.

[28] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, pp. 1157–1182, Mar. 2003, ISSN: 1532-4435. [Online]. Available: https://api.semanticscholar.org/CorpusID:379259.

[29] S. Shalev-Shwartz and S. Ben-David, "Understanding machine learning: From theory to algorithms," in Cambridge University Press, 2014, ch. Decision Trees.

[30] C. Strobl, A.-L. Boulesteix, and T. Augustin, "Unbiased split selection for classification trees based on the gini index," *Computational Statistics & Data Analysis*, vol. 52, no. 1, pp. 483–501, 2007, ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2006.12.030. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167947306005093.

[31] O. Khattab, M. Hammoud, and T. Elsayed, "Finding the best of both worlds: Faster and more robust top-k document retrieval," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '20, Virtual Event, China: Association for Computing Machinery, 2020, pp. 1031–1040, ISBN: 9781450380164. DOI: 10.1145/3397271.3401076. [Online]. Available: https://doi.org/10.1145/3397271.3401076.

[32] A. Trotman, A. Puurula, and B. Burgess, "Improvements to bm25 and language models examined," *Proceedings of the 19th Australasian Document Computing Symposium*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:207220720.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel, "PISA: performant indexes and search for academia," in *Proceedings of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, OSIRRC@SIGIR 2019, Paris, France, July 25, 2019.*, 2019, pp. 50–56. [Online]. Available: http://ceur-ws.org/Vol-2409/docker08.pdf.