Efficient Query Processing for Scalable Web Search

Nicola Tonellotto
National Research Council of Italy
nicola.tonellotto@isti.cnr.it

Craig Macdonald
University of Glasgow
craig.macdonald@glasgow.ac.uk

Iadh Ounis
University of Glasgow
iadh.ounis@glasgow.ac.uk

November 23, 2018

Abstract

Search engines are exceptionally important tools for accessing information in today's world. In satisfying the information needs of millions of users, the effectiveness (the quality of the search results) and the efficiency (the speed at which the results are returned to the users) of a search engine are two goals that form a natural trade-off, as techniques that improve the effectiveness of the search engine can also make it less efficient. Meanwhile, search engines continue to rapidly evolve, with larger indexes, more complex retrieval strategies and growing query volumes. Hence, there is a need for the development of efficient query processing infrastructures that make appropriate sacrifices in effectiveness in order to make gains in efficiency. This survey comprehensively reviews the foundations of search engines, from index layouts to basic term-at-a-time (TAAT) and document-at-a-time (DAAT) query processing strategies, while also providing the latest trends in the literature in efficient query processing, including the coherent and systematic reviews of techniques such as dynamic pruning and impact-sorted posting lists as well as their variants and optimisations. Our explanations of query processing strategies, for instance the WAND and BMW dynamic pruning algorithms, are presented with illustrative figures showing how the processing state changes as the algorithms progress. Moreover, acknowledging the recent trends in applying a cascading infrastructure within search systems, this survey describes techniques for efficiently integrating effective learned models, such as those obtained from learning-to-rank techniques. The survey also covers the selective application of query processing techniques, often achieved by predicting the response times of the search engine (known as query efficiency prediction), and making per-query tradeoffs between efficiency and effectiveness to ensure that the required retrieval speed targets can be met. Finally, the survey concludes with a summary of open directions in efficient search infrastructures, namely the use of signatures, real-time, energy-efficient and modern hardware & software architectures.

Acronyms

Here we report the main acronyms used in this survey. Acronyms typeset in Sans-serif pertain directly to information retrieval concepts that we explain in this survey.

NDCG	Normalised Discounted Cumulative Gain
MAP	Mean Average Precision
ERR	Expected Reciprocal Rank
MED	Maximised Effectiveness Difference
RBP	Rank Biased Precision
IR	Information Retrieval
QPS	Queries per second
IDF	Inverse Document Frequency
FOR	Frame-Of-Reference
PFOR	Patched FOR
Vbyte	Variable Byte
EF	Elias-Fano
PEF	Paritioned EF
QMX	Quantities, Multiplier and eXtractor
SIMD	Single Instruction Multiple Data
TAAT	Term-At-A-Time
DAAT	Document-At-A-Time
WAND	Weighted AND or Weak AND
BMW	Block-Max WAND
BMM	Block-Max MaxScore
BMA	Block-Max AND
LBMW	Local BMW
VBMW	Variable BMW
QEP	Query Efficiency Prediction/Predictor
QPP	Query Performance Prediction/Predictor
SLA	Service Level Agreement
PESOS	Predictive Energy Saving Online Scheduling
DVFS	Dynamic Voltage and Frequency Scaling
TFIDF	Term Frequency - Inverse Document Frequency
SAAT	Score-At-A-Time
LTR	learning-to-rank
FPGA	Field Programmable Gate Array
IoT	Internet-of-Things
ISN	index serving node

Notations

Here we only report the recurrent notation symbols used in this survey. `Fixed size` text is used for pseudocode-related symbols.

R	the number of replicas of a shard.
S	the number of shards of an index.
K	the number of top results returned by a search engine.
s	speedup, used as a performance measure.
r	reduction, used as a performance measure.
d	a document, as indexed by an IR system.
q	a query, as processed by an IR system, i.e., a set of terms.
N	the number of documents indexed by the IR system.
t, t_i	a term, as may exist within a query.
$\textsc{Score}_q(d)$	a generic query-document ranking function.
$s_t(q, d)$	a generic term-document similarity function.
f_t	the document frequency of a term.
IDF_t	the inverse document frequency of a term.
$f_{d,t}$	the number of occurrences of a term in a document.
$\perp$	special symbol to denote the end of a posting list.
n	number of terms in a query.
`p`, `I`, `O`	an array of posting lists.
`q`	a priority queue of docids or $\langle$`docid`, `score`$\rangle$ pairs.
`A`	an accumulators map from docids to scores.
λ	the size of an unordered window complex operator.
p	parallelism degree, i.e., number of threads.
$t(p)$	expected query processing time with p threads.
$\sigma_t(q)$	the term upper bound, a.k.a. its max score.
$\sigma_d(q)$, $\mathtt{\sigma_d}$	the document upper bound computed with term upper bounds.
$\hat{q}$	a set of terms from query q already processed.
θ, $\mathtt{\theta}$	a threshold, i.e., the smallest (partial) score of the current top K documents.
L	parameter of the Quit and Continue strategies.
N_t	the number of documents indexed in a top candidates list.
$\mathtt{\sigma}$	an array of term upper bounds.
`ub`	an array of document upper bounds.
`pivot`	a index of a posting list in `p`.
`pivot_id`	the docid of the `pivot` posting list iterator.
F	the aggressiveness tradeoff of a dynamic pruning strategy.
`b`	an array of block lists.

$\sigma_{\mathtt{b}}$	a document upper bound computed with block upper bounds.
$s_j(t)$	a QEP term statistic.
A_i	a QEP aggregation function, e.g., max, sum, variance.
$f_{ij}(q)$	a QEP feature defined for query q.
τ	a posting list score threshold.
$A(t_1, t_2)$	size of the intersection of the posting lists of terms t_1 and t_2.
δ, ϵ, β	small positive constants.
f_{ins}, f_{add}	filtering thresholds.
w	a weight.
b	the number of bits used to represent impacts.
g, h	fitting parameters for estimation of b.
L, U	global lower and upper bounds on term scores.
Q	a fidelity control knob.
ρ	number of postings to process.
M	a standard evaluation metric.
f_i, $\mathtt{f}$	a feature id.
$\mathcal{F}$	a feature id set.
$\mathbf{x}$	a feature vector.
T_i	a regression/decision tree.
$\mathcal{T}$	a set of regression/decision trees.
w_i	the weight of a regression/decision tree.
d_T	the depth of tree T.
s_i	a tree score contribution.
n_i	a branching node of a regression/decision tree.
$e_i(\mathbf{x})$	the exit leaf of a tree for a given feature vector.
$\mathcal{N}_i$	the set of branching nodes of a tree.
$\mathcal{L}_i$	the set of leaf nodes of a tree.
γ	a threshold value.
a	a array of two elements.
`tid`	an array of tree ids.
`mask`	an array of mask bitvectors.
`th`	an array of threshold values.
`exit`	an array of leaf bitvectors.
`scores`	a lookup table of scores.
$\mathcal{Q}$	a set of queries.

1 Introduction

Search engines are exceptionally important tools for accessing information in today's increasingly digital world. Classical commercial Web search engines, such as those maintained by Google, Bing, Yandex, Baidu, have processed billions if not trillions of Web documents, and have kept maintaining these in continuously updated index data structures[1] requiring

[1] https://www.google.com/insidesearch/howsearchworks/thestory/index.html

petabytes of storage space,[2] to ensure *satisfying* the users of the search engine through billions of user queries received every month.[3] (Bosch *et al.*, 2016)

Satisfaction of the search engine users is a key metric for search engine providers. Without drawing too broad a sweeping generalisation, one of the fundamental goals of a search engine is to derive income from advertising traffic, for instance from the ads that are often presented next to the organic search results. Users that are not satisfied with the search engine results may switch to a different engine (White, 2016), and may not return. This is a loss of advertising revenue for the search engine. As a consequence, ensuring that their users are satisfied with the results is of utmost importance to search engines.

There are various reasons why the result page for a search does not satisfy a user (Diriye *et al.*, 2012), but the primary causes are the *effectiveness* – the quality of the returned results – and the *efficiency* – the speed at which the results were returned. Indeed, search engines that are slow to return results to users can negatively damage the user's perception of the quality of the results (Brutlag and Schuman, 2009). Hence, a search engine needs to be both effective (deploying advanced ranking mechanisms), while ensuring that its results are efficiently returned to the user. A key contribution of this survey is to review both the foundational background and the recent advances in search engine infrastructures.

Fortunately, search is a parallelisable problem, and scaling can be applied to the search engine computing infrastructure. Indeed, as shown in Figure 1.1, a large index can be partitioned across multiple *shards*, allowing each single search engine server to service a small portion of the index in order to ensure fast retrieval. Each of the S index shards can be replicated R times, allowing both resilience and scaling. When user queries arrive, the *broker* routes queries to the less loaded replica of each shard for processing (Freire *et al.*, 2013; Freire *et al.*, 2012).

Using such a distributed setting for a large search engine, $R \times S$ can be very large, covering potentially hundreds of thousands of servers. All of the major search engines run exceedingly large data centres, each often requiring capital investments of billions of dollars,[4] and consuming vast quantities of energy. Data centres use 3% of the global electricity supply and account for about 2% of the total greenhouse gas emissions; this is expected to triple in the next decade, putting an enormous strain on energy supplies and dealing a hefty blow to efforts to contain global warming.[5]

Clearly, at such large scales, the efficiency of the search engine's operating internals are therefore key to the operating costs of such companies. Efficiency improvements of 5% would allow a 5% reduction in R replicas, potentially equating to significant power

[2] https://www.google.com/insidesearch/howsearchworks/crawling-indexing.html

[3] https://googleblog.blogspot.it/2010/09/google-instant-behind-scenes.html

[4] http://www.datacenterknowledge.com/archives/2014/07/23/from-112-servers-to-5b-spent-on-google-data-centers-per-quarter

[5] http://www.independent.co.uk/environment/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html

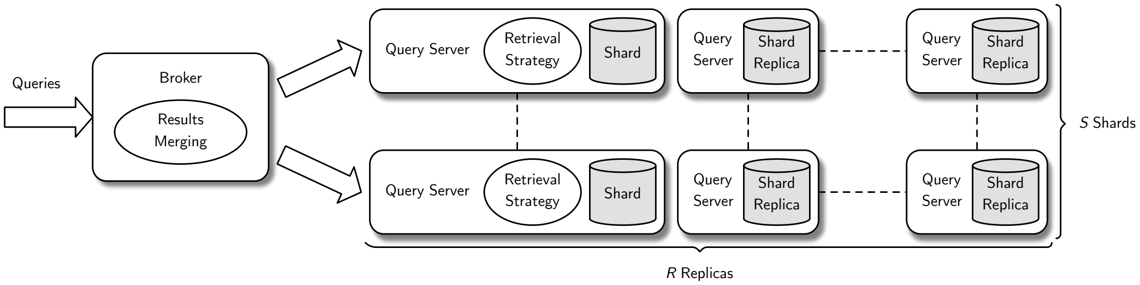


Figure 1.1: Distributed retrieval architecture.

consumption reductions, and providing a room for further growing the sizes of the search engines' indexes, or servicing growth in the user queries.

The distributed nature of a search engine infrastructure is not within the scope of this survey. The interested reader can find a comprehensive overview of distributed large-scale Web search engines in (Cambazoglu and Baeza-Yates, 2015). Instead, this survey focuses on the general architecture of a search infrastructure as might be deployed within a single server. Our goal is to provide an accurate description of the basic search components involved in the scoring of documents in response to a query, together with a detailed and exhaustive review of the research works aiming at boosting the efficiency of query processing without negatively impacting the effectiveness performance of the system.

A key detail of the manner in which a search engine is designed to operate is the "top-heavy nature" of results: since the users of search engines typically focus on the top-ranked results (as can be measured offline using test collections and metrics such as NDCG (Järvelin and Kekäläinen, 2002) and ERR (Chapelle *et al.*, 2009)), the relevance of those results is key to user satisfaction. This means that the search engine should itself focus on getting the most relevant results at the top of the ranking, at the possible detriment of mis-ranking other results. In his SIGIR 2010 Industry Day talk, Pedersen (2010) described this process as the use of cascading (illustrated in Figure 1.2). In response to a query, each conceptual cascade aims to filter or rank documents, before passing onto the next cascade layer. At the bottom layer, the documents to be retrieved are defined in terms of the subsets of terms present in the query – being able to identify these subsets as quickly as possible, without requiring to scan the contents of each document, is a fundamental architecture decision of an Information Retrieval (IR) system. The bottom layer may filter a collection of billions of documents down to the millions, which should be scored. In the second layer, query processing techniques define how the scoring of document weighting models, such as language modelling or BM25 should be applied. In the final layer (the top layer), various additional ranking features such as PageRank, or URL information may be calculated and used within a learned model to re-rank the documents, before presenting the final top

K high-scored documents to the user (usually K is small, e.g., $8 - 20$, as displayed on the first page of the search results).

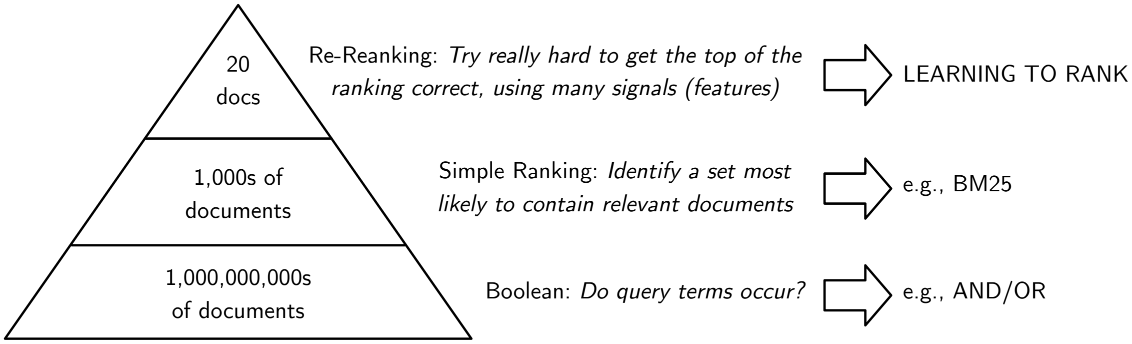

20
docs
1,000s of
documents
1,000,000,000s
of documents
Re-Reanking: *Try really hard to get the top of the ranking correct, using many signals (features)*
LEARNING TO RANK
Simple Ranking: *Identify a set most likely to contain relevant documents*
e.g., BM25
Boolean: *Do query terms occur?*
e.g., AND/OR

Figure 1.2: Cascading nature of Web search, based on (Pedersen, 2010)

Different techniques are appropriate at different cascade levels, but many are designed to make efficiency savings by avoiding the scoring of documents, which cannot make the top-ranked results that will be returned to the user. In this survey, we cover both the core algorithms and data structures used for retrieval, as well as the optimisations (such as dynamic pruning) that can be applied at a given cascade level. Of course, not all queries are equal – some are easier for the search engine to answer *effectively*, while, orthogonally, some may be less *efficient*, i.e., take longer for the search engine to answer. Being able to know the likely efficiency of a query, as might be obtained from a *query efficiency predictor*, can allow the search engine to make on-the-fly decisions about its configuration.

Figure 1.3 provides the main infrastructure that is discussed in this survey. We will focus on the "online" components, e.g., those responsible for the cascading components of search, while referring to the "offline" components whenever it is necessary. The remainder of this survey is structured as follows:

- Chapter 2 provides an overview of the modern infrastructure foundations within a search engine, covering the basic form of the inverted index data structure, and the essentials of query processing.
- Chapter 3 provides an introduction to approaches for increasing the efficiency of query processing, namely the dynamic pruning techniques.
- Chapter 4 describes query efficiency predictors – a new technique to estimate the response time of queries – that is gaining attention for a number of applications involving efficient retrieval on a per-query basis.
- Chapter 5 provides an overview of impact-sorted indexes, which make offline changes to the layout of the inverted index in order to improve the efficiency of query processing.
- Chapter 6 provides an overview of cascading search architectures, and provides insights into how to efficiently deploy learning-to-rank, a retrieval technique known to benefit

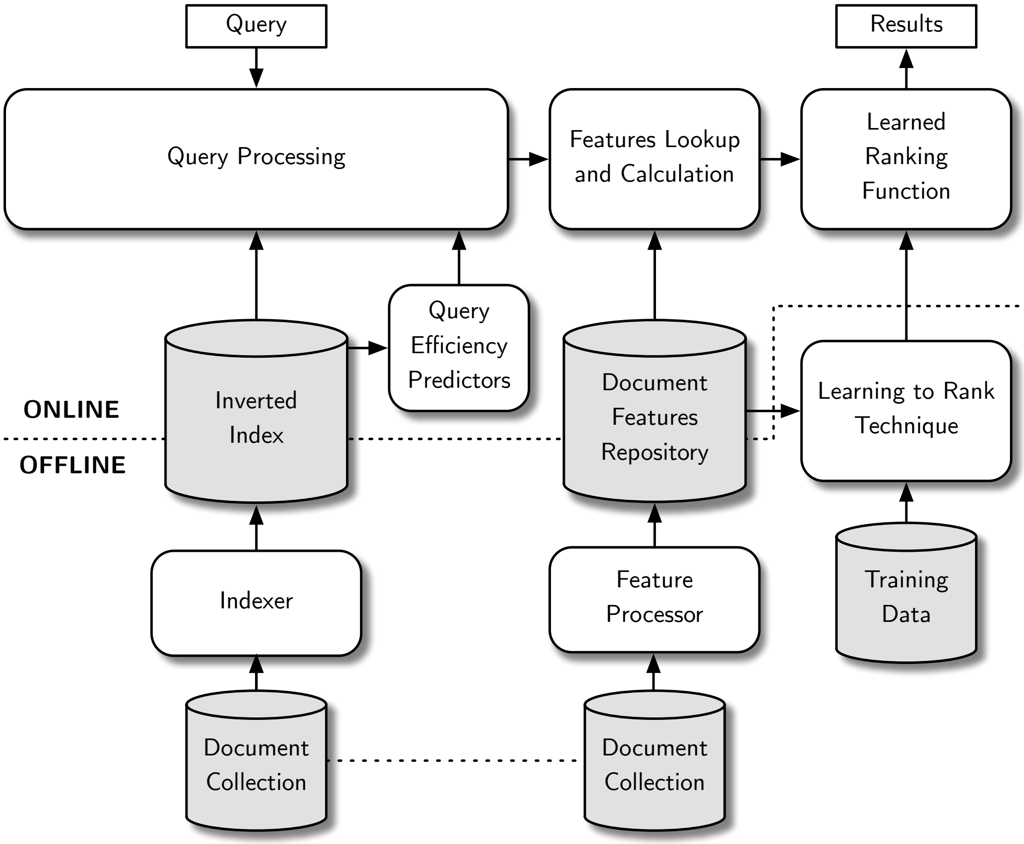


Figure 1.3: A conceptual architecture for a search engine.

the search engine's effectiveness by re-ranking a set of K documents.

- Chapter 7 gives an overview of the current open directions in retrieval infrastructures, including the use of signature files instead of inverted indexes, and provides concluding remarks.

Note on Efficiency Performance Measures

In this survey, we illustrate the efficiency measures reported in the cited papers. Since this survey covers papers from over a period of 30 years, comparing the reported results across different papers could lead to the wrong conclusions. Hence, we will only report comparative performance measures derived from single contributions.

The performances of the discussed strategies naturally depend on several factors, such as the index and/or the query characteristics, the inverted index compression, the similarity function adopted, the number of documents returned, the actual underlying implementations, the machine(s) used to perform the experiments and so on. In most papers, when comparing the efficiency of different solution s, two main quantities are typically reported: response times and/or number of processed elements. In order to be as "implementation-independent" as possible, we report the speedup of an optimisation w.r.t. the baseline, in terms of mean response time, and/or its (work) *reduction*, defined as the percentage of postings that are dynamically pruned, i.e., not scored, w.r.t. the baseline.

When comparing two time quantities t_1 and t_2, with $t_1 > t_2$ we will always report their relative *speedup* s, defined as $s = t_1/t_2$ (always greater than 1). For example, if two strategies A and B have an average response time of 20 ms and 8 ms, respectively, their speedup (of B w.r.t. to A) is $s = t_A/t_B = 20/8 = 2.5\times$. When comparing two numbers of processed elements n_1 and n_2, with $n_1 > n_2$ we will systematically report the percentage *reduction* r, defined as $r = 1 - n_2/n_1$. For example, if strategy A processes 200 elements while strategy B processes just 150 elements, the reduction of B w.r.t. A is $r = 1 - n_B/n_A = 1 - 150/200 = 0.25 = 25\%$.

Finally, the throughput of a query processing node, as well as that of more complex search systems, is measured in *queries per second* (QPS).

Intended Audience

This survey targets readers, researchers and engineers who possess a basic knowledge in Information Retrieval (IR) or in other cognate topics (e.g., databases, data mining). In particular, the survey is of utmost interest to PhD students, researchers and practitioners working on efficiency and system infrastructures in IR and Web search. Indeed, anyone working on search and ranking on big data will benefit from this manuscript. The survey is also particularly of interest to lecturers and tutors looking for a concise and comprehensive textbook on state-of-the-art query processing techniques to support their IR course.

Note on the Origins of the Material

This survey is a new piece of work, but builds upon our research experience in this area. This survey also benefits from the authors' experience acquired from presenting two related tutorials at ECIR 2017 and SIGIR 2018. We would like to thank the attendees of these tutorials for their insightful questions and comments.

2 Modern Infrastructure Foundations

In this chapter we focus on the fundamental concepts that will be needed in the rest of the survey. Indeed, we provide a general description of the main "ingredients" that are necessary to later introduce the more advanced components of query processing. These fundamental ingredients include, in Section 2.1, the nature of the data structures underlying an IR system – namely the vocabulary and inverted index – and how the inverted index can be compressed to reduce space usage and decompression time; in Section 2.2 we describe the basic query processing algorithms that permit retrieval from an inverted index using both Boolean and best-match retrieval.

2.1 Data Structures

The goal of an IR system is to return information objects relevant to a user's information need, expressed as a query. We will refer to the information objects as *documents*, and the set of documents over which we perform information retrieval as the *document collection*. Each document d can be uniquely identified by a natural number $0, 1, \ldots, N-1$, which is called a *document identifier* or *docid*. The atomic unit of information that can be observed or extracted from a document is called a *token*. Tokens represent occurrences of single *terms*, and the set of unique terms from all documents in a collection is called the *vocabulary* or *lexicon*. Terms can also be uniquely identified with a natural number, called a *term identifier* or *termid*.

The most common type of documents managed by an IR system are the textual documents, such as text files or Web pages. Hence, the tokens are the occurrences of words in a document, and a word corresponds to a term. A user query q is expressed as a bag of terms, and an IR system exploits such terms to process the query and to select documents from the collection that can satisfy the user's information need. Since one of the main goals of an IR system is to provide these relevant documents with subsecond response times, the document collection must be organised and stored in special data structures to quickly locate data without processing every document in the collection.

An *inverted index* (also known as an *inverted file*) is the most efficient data structure for accomplishing the goal of text query processing (Witten *et al.*, 1999; Zobel and Moffat, 2006; Manning *et al.*, 2008; Baeza-Yates and Ribeiro-Neto, 2008; Croft *et al.*, 2009; Büttcher *et al.*, 2010).[1] An inverted index organises a document collection in a collection of lists, one per term (or token), containing information about the documents in which the term appears.

An inverted index encodes a document collection, exploiting different data structures that depend on the used query processing algorithm. A typical inverted index layout is depicted in Figure 2.1. In any case, two components are always present: (a) the *vocabulary* (or *lexicon*) and (b) the set of *posting lists* (or *inverted lists*).

The *vocabulary* is an array of term entries. Every term entry stores information about

[1] Alternative data structures include signature files and suffix arrays (Baeza-Yates and Ribeiro-Neto, 2008)

a term in the document collection, and, in particular, its *document frequency* (docfreq), i.e., the number of documents in which the term t appears at least once, and a *pointer* to the beginning of the posting list of term t. Note that a vocabulary may store additional information (stats) regarding the terms appearing in the documents (e.g., the total number of occurrences of the term in the collection). Typical implementations of a vocabulary include front-coded sorted arrays, tries, hash tables or B-like trees to allow reduced search time and an early lookup of term entries (Baeza-Yates and Ribeiro-Neto, 2008).

A *posting list* is associated with every term in the vocabulary, and it is composed of a list of *postings*. A posting is a logical representation of all the information extracted from a single document in the collection about a specific term. In the most basic case, a posting for a given document d and term t contains the docid associated with d and, as such, the posting list of term t contains the docids of the documents in the collection in which the term appears. Other information, such as the frequency of the term's occurrence in each document (freq), or the positions of its occurrences (positions) can also be encoded in the posting.

Finally, an optional component of an inverted index is the *document index*. A document index stores the different attributes of documents as an array of records, indexed by docid. These attributes include statistical properties of documents, such as the total number of words (i.e., its *document length* doclen) and other properties props, such as the language and the number of incoming links to the document (inlinks), as well as other pre-computed attributes, such as the importance of the document (i.e., its global score).

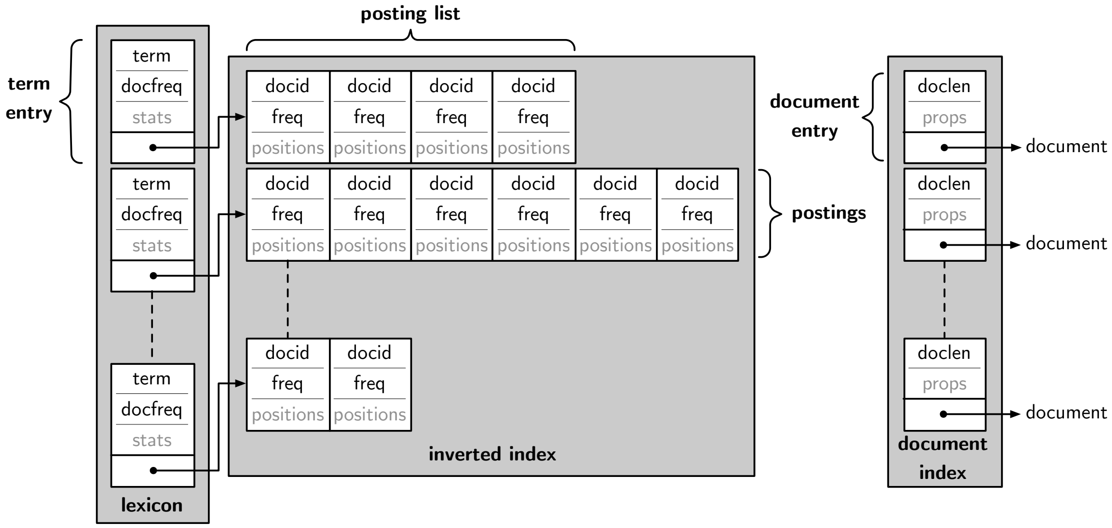


Figure 2.1: A typical inverted index components. Greyed-out elements are typically optional, and depend on the choices made in the implementation of the search engine.

Overall, the most important data structure is the *inverted index*, whose main functions and characteristics are described in Section 2.1.1. Moreover, the inverted index is normally stored on disk, but since it must be accessed repeatedly to process queries, it is often kept in the main memory to reduce costly disk accesses. Since an inverted index can occupy a large amount of space, compression techniques are employed to reduce its size (without loss of information). Several such compression techniques are described in Section 2.1.2. In contrast, the "lossy" compression of the inverted index – known as static index pruning – is highlighted later in Section 5.

2.1.1 Accessing an Inverted Index

The inverted index makes it possible to identify the documents in which some terms appear without analysing all documents in the collection. By selecting the posting lists of the query terms, a query processing algorithm can traverse them to identify the docids of documents in which at least one (or all) of the terms appear(s). Commercial Web search engines often internally rewrite the user query into a complex query plan that includes boolean expressions of query terms, which must then be resolved (Risvik *et al.*, 2013). This posting list query processing is known as *boolean retrieval* (see Section 2.2.1). In order to reduce their space occupancy, to store more documents in the same amount of space, and to exploit modern memory hierarchies, for faster access, the posting lists are stored contiguously, traversed sequentially, and are usually compressed (see Section 2.1.2) (Zobel *et al.*, 1998). Large compression benefits can be attained if the posting lists are sorted by increasing docid value (Silvestri, 2007).[2]

An inverted index with posting lists sorted by increasing docid value is called a *docid-sorted* inverted index. This index layout is commonly used in Web search engines (Dean, 2009). Other index layouts exist, for example the score-sorted or impact-sorted indexes, which are discussed in chapter 5. Unless otherwise specified, we will always assume that the posting lists are sorted by increasing docids.

Boolean query processing on Web-scale document collections, with billions of texts to be managed, cannot be used to retrieve all matching documents directly for the users. Users do not typically browse more than 20 results (Silverstein *et al.*, 1999). As a consequence, the number of documents to return to users must be often limited. Thus, a *relevance score* is associated with the query-document pairs. Documents are ranked according to a heuristic similarity function, estimating, according to a given statistical procedure, the similarity (or the probability of relevance) of a document with respect to a query. Then, the documents in the posting lists are sorted by their similarity to the user query, and the K documents with the highest scores are returned to the user. This posting list processing is known as *ranked retrieval* (see Section 2.2.2). Sometimes, the final similarity of a document with

[2] Different orderings, such as impact-based, can lead to greater compression benefits (see Chapter 5).

respect to a query is modified by taking into account the query-independent features of the document (e.g., PageRank, URL length). These features provide the *static* or *global score* of a given document. The effective integration of many such features raises several challenges with respect to their weighting, often addressed by the use of learning-to-rank techniques, further discussed in Chapter 6.

Many query-document similarity functions have been proposed, including: the cosine measure (Salton *et al.*, 1975), BM25 (Robertson *et al.*, 1994), statistical language measures (Ponte and Croft, 1998; Lafferty and Zhai, 2001), and divergence from randomness measures (Amati and Van Rijsbergen, 2002). We will not enter into the details of specific similarity measures, but we will compute the relevance score of query-document pairs through a generic ranking function $\textsc{Score}_q(d)$ following the general outline given by the best match strategy, namely:

$$\textsc{Score}_q(d) = \sum_{t \in q} s_t(q, d) \tag{2.1}$$

where $s_t(q, d)$ is a term-document similarity function that depends on the number of occurrences of term t in document d (i.e., the within-document term frequency) and in the query q (i.e., the within-query term frequency), on other document statistics such as document length and on term statistics such as the document frequency. Commonly, the document frequency of a term f_t is used in the similarity functions through a quantity called *inverse document frequency* IDF_t,[3] denoted as:

$$\mathsf{IDF}_t = \log \frac{N}{f_t} \tag{2.2}$$

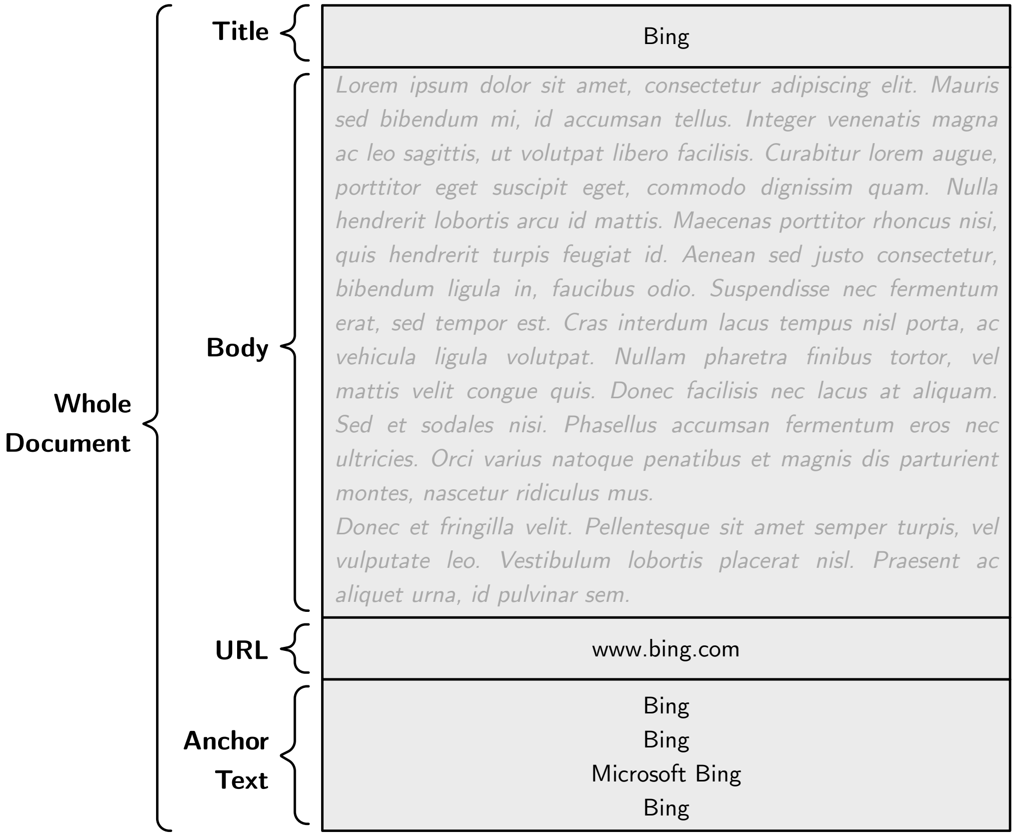
where N is the number of documents in the collection. As such, rare terms will have a high IDF_t, and a high similarity score, while common terms will likely have a low IDF_t and a low similarity score. Term-document similarities are non-negative quantities, and the contribution to the similarity of document terms not appearing in the query is assumed to be 0, and vice-versa.[4] Since the in-document term frequency is commonly used in many similarity measures, it is a typically included in the posting list (freq).

An inverted index storing posting lists as sequences of $(d, f_{d,t})$ pairs is called a *document-level* index, since each posting contains information about the number of occurrences of a term in a document $f_{d,t}$, but no finer-grained information on these occurrences is provided. Alternatively, a *term-level* index encodes in each posting additional information on the position of the occurrences of a term in a document. In particular, each posting stores an

[3] For the same purpose, some weighting models, such as the language models and divergence from randomness models, rely upon the total frequency of the term in the entire collection.

[4] More generally, the query-document similarity may also include an additive query independent document global score $G(d)$.

increasing sequence of natural numbers, encoding the position of each occurrence of term t in document d (positions). In this way, it is possible to establish if two or more terms have adjacently occurred or not.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris sed bibendum mi, id accumsan tellus. Integer venenatis magna ac leo sagittis, ut volutpat libero facilisis. Curabitur lorem augue, porttitor eget suscipit eget, commodo dignissim quam. Nulla hendrerit lobortis arcu id mattis. Maecenas porttitor rhoncus nisi, quis hendrerit turpis feugiat id. Aenean sed justo consectetur, bibendum ligula in, faucibus odio. Suspendisse nec fermentum erat, sed tempor est. Cras interdum lacus tempus nisl porta, ac vehicula ligula volutpat. Nullam pharetra finibus tortor, vel mattis velit congue quis. Donec facilisis nec lacus at aliquam. Sed et sodales nisi. Phasellus accumsan fermentum eros nec ultricies. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Donec et fringilla velit. Pellentesque sit amet semper turpis, vel vulputate leo. Vestibulum lobortis placerat nisl. Praesent ac aliquet urna, id pulvinar sem.

Figure 2.2: Different fields within a document, from (Macdonald *et al.*, 2013b).

Finally, it is worth noting that a given document can be represented in different ways. For example, a Web page can be represented by its URL, its title or its content. Different document representations entail different term and term-document statistics, e.g., the term frequency in a Web page title typically differs from the frequency of the same term in the content. Hence, an inverted index could group together the term statistics and the corresponding posting lists for different document representations, also known as *fields*. As shown in Figure 2.2, the typical fields in a Web page document are the URL, title, content and anchor text from the incoming hyperlinks (Macdonald *et al.*, 2013b).

Physically, the internal organisation of posting lists can be *interleaved* or *non-interleaved*, as shown in Figure 2.3 (Anh and Moffat, 2006c). In interleaved indexing, each posting is stored compactly, where the docid is immediately followed by the term-document frequency and, for term-level indexes, by the list of term occurrence positions. In a non-interleaved index, all individual components of postings are stored in different (portions of the) inverted files: one for the docids, one for the frequencies and one for the term positions.

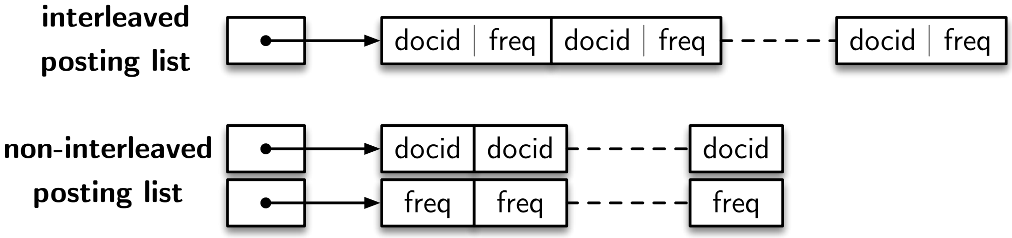


Figure 2.3: Interleaved and non-interleaved posting list organisations.

Since posting lists are processed sequentially, it is often convenient to see a posting list as an *iterator* over its postings. Hence, the access of a posting list through its pointer returns an iterator starting on its first posting. Some operations that are commonly performed on a posting list `plist` are defined as follows:

- `plist.docid()` returns the document identifier d of the current posting. If the iterator has reached the end of the posting list, `plist.docid()` returns the special symbol $\perp$. For comparison purposes, the special symbol $\perp$ is considered strictly greater than any other docid.
- `plist.score()` returns the similarity score computed with the term and document statistics extracted from the current posting, as calculated by $s_t(q, d)$ in Equation (2.1).
- `plist.next()` sequentially moves the iterator to the next posting. If the iterator has reached the end of the posting list, `plist.next()` returns the special symbol $\perp$, the end-of-list marker.
- `plist.next(`d`)` advances the iterator forward to the next posting with a document identifier greater than or equal to d.[5] If the current posting's docid is greater or equal to d, the iterator is left unchanged. If d is greater than the docid of the last posting in the list, `plist.next(`d`)` returns the end-of-list marker $\perp$. In Section 2.1.2, we will discuss the efficient implementation of the `plist.next(`d`)` operator, such that the number of intervening postings between the current document and the target d that are decompressed is minimised.

These operations, also known as *posting APIs*, were introduced by (Broder *et al.*, 2003). The `plist.next()` and `plist.next(`d`)` operators advance sequentially the iterator along a posting list and, when required, the `plist.docid()` and `plist.score()` operators are used to inspect and process the contents of posting currently identified by the iterator. As we will see in Section 2.2, the posting list APIs play a fundamental role in all query

[5] Alternatively, this operation can be denoted as `plist.nextGEQ(`d`)`.

processing strategies.

2.1.2 Compression & Skipping

Compression is a fundamental component for query processing, since its main aim is to reduce the space required to store the inverted index while providing acceptable latencies when using it for query processing. The compression of the posting lists within the inverted index ensures that as much as possible of the inverted index can be kept in the higher levels of the computer memory hierarchy – indeed, many search engines keep the entire inverted index in the main memory. Hence, a compression scheme should not only be time-efficient (i.e., inexpensive to decompress), but also space-efficient (i.e., high compression ratio), to minimise the necessary computing resources while answering queries. Inverted index compression has been used for some time. For example, one common practice while storing a posting list is to use *gaps* (or *d-gaps*) where possible (Witten *et al.*, 1999), i.e., to record the differences between components (such as docids or positions) instead of their absolute actual values. Gaps between docids are expected to be small, requiring far less space to store than the complete docids. Indeed, by not using a fixed (word-aligned) number of bits for each number, smaller numbers generally lead to smaller representations in terms of bits. In the following, we use the term *codec* to describe a compression/decompression algorithm, and recap the basic (also called *oblivious*) compression codecs, (i.e., Unary, Gamma, Vbyte and Varint) as well as list-adaptive codecs (i.e., Golomb/Rice, Simple, Frame-Of-Reference, and Elias-Fano). We then discuss some recent advances in compression leveraging the modern processor architectures. Finally, we discuss docid assignment and the efficient implementation of skipping, as needed by query processing strategies.

Oblivious Codecs

When a set of non-negative integers is given to an oblivious codec for compression, it encodes each value on its own, without considering its value relative to the rest of the set. A desirable side effect is that every single value can be decompressed separately, or only the decompression of the preceding values is needed if d-gaps are used. On the other hand, such codecs ignore global information about the set, which can help to have a better compression ratio. A number of oblivious compression algorithms are briefly described below.

Unary and Gamma: Unary and Gamma codecs are two bitwise, oblivious techniques. Unary represents a non-negative integer x as $x - 1$ one bits and a zero bit (e.g.: 4 is 1110). While this can lead to extremely large representations, it is still advantageous for the encoding of values that tend to be small, such as those created by the application of delta gaps, or term frequencies in small document collections. Gamma, described in (Elias, 1975), represents positive integer x as the Unary representation of $1 + \lfloor \log_2 x \rfloor$ followed by the binary representation of $x - 2^{\lfloor \log_2 x \rfloor}$. (e.g.: 9 is 1110 001).

Vbyte: Vbyte (variable byte) codec (Williams and Zobel, 1999) is a byte-aligned, oblivious technique. It uses the 7 lower bits of any byte to store a partial binary representation of the non-negative integer x. It then marks the highest bit as 0 if another byte is needed to complete the representation, or as 1 if the representation is complete. For example, 201 is 10000001 01001001. While this technique may lead to larger representations, it is usually faster than Gamma in terms of decompression speed (Scholer *et al.*, 2002). Trotman (2014) noted that there are different possible implementations of Vbyte, including the Group Varintused by Google (Dean, 2009).

List-adaptive Codecs

A list-adaptive codec compresses non-negative integers in blocks, exploiting aspects such as the proximity of values in the compressed set. This information can be used to improve the compression ratio and/or decompression speed. However, this can mean that an entire block must be decompressed even when just a single posting is required from it (e.g. during partial *dynamic* scoring approaches such as WAND, as described in chapter 3). Moreover, it is possible to obtain a larger output than the input when there are very few integers to compress within the block, because extra space is required in the output to store the header information needed at decompression time (e.g., range of integers, number of bits per integer). Indeed, when there are too few integers to be compressed, this header information can be larger in size than the actual payload being compressed (or the inputs have to be padded with superfluous extra integers). Below, we provide short descriptions of common list-adaptive techniques.

Golomb and Rice: Golomb codec is a bitwise and list-adaptive compression scheme (Golomb, 1966). Here, a non-negative integer x is divided by a value β. Then, Unary codec is used to store the quotient q while the remainder r is stored in binary form. The value of β is chosen depending on the non-negative integers we are compressing. Usually, $\beta = 0.69 \times \mathsf{avg}$ where avg is the average value of the numbers being compressed (Witten *et al.*, 1999). In the Rice codec (Rice and Plaunt, 1971), β is a power of two, which means that the bitwise operators can be exploited, permitting more efficient implementations at the cost of a small increase in the size of the compressed data. Golomb and Rice coding are well-known for their decompression inefficiency (Anh and Moffat, 2005a; Yan *et al.*, 2009a; Lemire and Boytsov, 2015).

Simple family: This family of techniques, firstly described in (Anh and Moffat, 2005a), stores as many non-negative integers as possible in a single word. This is made possible by using the first 4 bits of a word as a *selector* to describe the organisation of the remaining 28 bits. For example, in a word we can store $\{509, 510, 511\}$ as three 9-bits values, with the highest 4 bits of the word reflecting this configuration, at a cost of one wasted bit.

Frame of reference (FOR): Proposed by Goldstein *et al.* (1998), FOR compresses non-negative integers in blocks of fixed size (128 elements, for example). It computes the range of the integers, i.e., between the maximum M and the minimum m value in the block ($M - m$), then stores m in binary notation. Each of the values are then saved, using their difference from m and encoded using b bits each, where $b = \lceil \log_2(M + 1 - m) \rceil$.

Patched frame of reference (PFOR): FOR may lead to a poor compression in presence of outliers: single large values that force an increase of the bit width b on all the other elements in the block. To mitigate this issue, PFOR has been proposed (Zukowski *et al.*, 2006). This approach chooses a b which is reasonable for most of the elements in the block, treating these as in FOR. The elements in a range larger than 2^b are treated as *exceptions*. In the original approach, those are stored at the end of the output, not compressed. The unused b bits are used to store the position of the next exception in the block. If b bits are not enough to store the position of the next exception, a compulsory one is generated. This means that one of the value is treated as an exception even if it could have been encoded using b bits.

More recent implementations of PFOR treat the exceptions differently. NewPFD (Yan *et al.*, 2009b) stores the exception positions in a dedicated area of its output, but divides them in two parts: b bits are normally stored, as for the normal values, while the remaining 32-b bits are compressed using a Simple family codec. OptPFD (Yan *et al.*, 2009b) works similarly, but chooses b in a way that tries to optimise the compression ratio and the decompression speed. FastPFOR (Lemire and Boytsov, 2015), instead, reserves 32 different areas of its output to store exceptions. Each area contains those exceptions, which can be encoded using the same number of bits. Outliers in the same exception area are then compressed using a FOR technique, to improve both the compression ratio and the decompression speed.

Elias-Fano (EF): Recently, the EF representation of monotone sequences (Elias, 1974; Fano, 1971) has been applied to inverted index compression (Vigna, 2013). Given a monotonically increasing sequence of n non-negative integers upper-bounded by u, each element in the sequence is represented in binary using $\lceil \log u \rceil$ bits. The binary representation of each element is split into two parts: a high part, consisting of the $\lceil \log n \rceil$ most significant bits, and a low part consisting of the remaining bits. The low parts are stored in a fixed-width bitvector, while the high parts are represented in negated unary.[6] Overall, it can be shown that the sequence representation takes $n \lceil \log \frac{u}{n} \rceil + 2n$ bits. Moreover, the EF representation of a compressed sequence can support random access without decompression by using an auxiliary succinct data structure, i.e., negligibly small compared to the EF space occupancy (Vigna, 2013). However, EF fails to exploit the local clustering that inverted lists usually exhibit, namely the presence of long subsequences of close document identifiers. More recently, (Ottaviano and Venturini, 2014) described a new representation based on partitioning the list into chunks and encoding both the chunks and their endpoints with

[6] Negate unary represents a non-negative integer x as $x - 1$ zero bits and a one bit.

EF, hence forming a two-level data structure, called Partitioned Elias-Fano (PEF). This partitioning enables the encoding to better adapt to the local statistics of the chunk, thus exploiting clustering and improving compression. They also showed how to minimise the space occupancy of this representation by setting up the partitioning as an instance of an optimisation problem, for which they present a linear time algorithm that is guaranteed to find a solution at most $(1 + \epsilon)$ times larger than the optimal one, for any given $\epsilon \in (0, 1)$.

SIMD-based Compression

Modern efforts in compression have increasingly focussed on the use of Single Instruction Multiple Data (SIMD) CPU instructions (Stepanov *et al.*, 2011), which permit the same operation on multiple data points simultaneously.

For example, Varint-G8IU (Lemire and Boytsov, 2015; Stepanov *et al.*, 2011; Trotman, 2014) is a form of Group Varint where as many integers are packed into 8 consecutive bytes preceded by a one-byte descriptor that depicts the number of encoded integers. A single CPU instruction, PSHUFB can then decode all integers.

Lemire and Boytsov (2015) provided a comprehensive study of SIMD-based compression codecs for integers. For instance, SIMD-BP128 packs blocks of 128 consecutive integers into a number of 128-bit words. Each integer is stored using the same number of bits. Then a single SIMD instruction can decode 16 128-bit words at once. The selector is stored before the sequence, and defines the number of bits needed for each integer.

QMX, which was proposed by Trotman (2014), builds upon SIMD-BP128, but separates the payload (or Quantities), the run length (or Multipliers), and the selector (or eXtractor), hence the name. QMX has been shown to be more space efficient than SIMD-BP128 (particularly for short posting lists), as well as faster at decoding on most of the CPU architectures investigated by Trotman (2014).

Docid Orderings

The compression rate achievable by a given codec depends on how the occurrences of terms are clustered within the docid space, due to the use of d-gaps: the smaller the d-gaps exhibited, the higher is the attainable compression.

Various works have examined the *assignment* of docids to documents to benefit compression – in this way, the collection is reordered, such that docid 0 is no longer the first document observed during indexing. Different schemes aim to achieve a clustering property – by clustering together similar documents, which contain similar terms – hence similar documents would obtain close docids, and hence the posting lists for these terms would have smaller d-gaps (Bookstein *et al.*, 1997). In general, finding such an ordering is NPhard (Silvestri, 2007), and hence various heuristics have been investigated in the literature (e.g., (Shieh *et al.*, 2003; Silvestri *et al.*, 2004; Tonellotto *et al.*, 2011)).

However, work in this area has been largely completed, due to the observations of Silvestri (2007), who demonstrated that by simply ordering documents lexicographically by their URLs (reversing the host part of the URLs first[7]) produces a good approximation of the clustering property.[8] Indeed, Silvestri (2007) observed compression ratios 5% better than a clustering approach, while taking two orders of magnitude less time to compute the docid reassignments. Later, Tonellotto *et al.* (2011) demonstrated the benefits of URL ordering on efficient retrieval approaches such as DAAT, MaxScore and WAND. More recently, Ramaswamy *et al.* (2017) showed that a docid reordering based on the item category reduces the average latency of over 20% in a large-scale eCommerce search system.

Nevertheless, major search engines, dealing with potentially a trillion documents, may find it expensive for the indexing subsystem to assign a globally unique docid to every document (Risvik *et al.*, 2013). While space compression helps in dealing with storage costs, distributed search engines may rely on a *random* docids assignment, to help load balancing across the index shards.

Skipping

When processing a posting list, there are numerous cases in which it is possible to avoid reading and decompressing portions of the list. For example, looking for a specific docid in a given posting list, and reading from disk and/or decompressing all preceding docids is just time consuming. Hence, (Moffat and Zobel, 1996) proposed to insert *synchronisation points* into a compressed posting lists. Such synchronisation points are stored in the index, together with the corresponding posting lists. Each of them is represented by a docid-offset pair, also known as a *skip pointer*, denoting the offset in bits/bytes w.r.t. the beginning of the posting list where the given docid is compressed (see Figure 2.4).

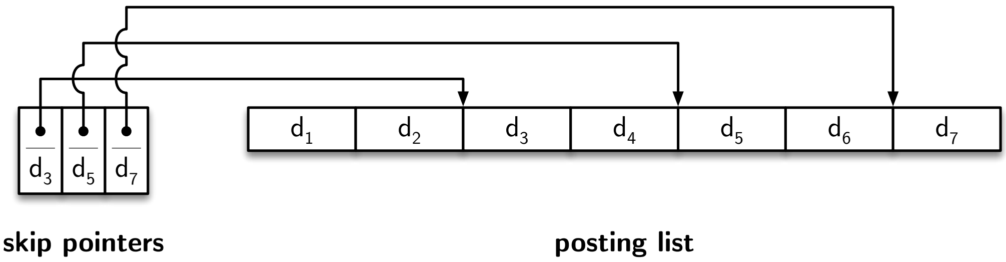


Figure 2.4: Example of three skip pointers in a posting list with seven postings.

[7] For example, www.example.com/main.html becomes com.example.www/main.html.

[8] Note that this technique will not work if all documents are from the same domain, e.g., Wikipedia.

To check if a document appears in the posting list we use the skip pointer with the greatest docid smaller than the document's docid, and we use its offset to access the inverted index starting from the position of such docid. In fact, skip pointers are used to efficiently implement the `plist.next(`d`)` operator, introduced at the end of Section 2.1.1. According to Moffat and Zobel (1996), skip pointers should be placed sequentially at fixed-width intervals, every $\sqrt{f_t}$ posting, where f_t represents the number of postings in the posting list of term t.

More recently, Chierichetti *et al.* (2008) proposed a dynamic programming approach to place skip pointers optimally, i.e., minimising the expected time to process a query, when a probability distribution of the processed terms is known in advance. (Boldi and Vigna, 2005) proposed to embed whole *skip lists* into the posting list for faster access, including multiple-level of skip pointers.[9] However, due to the prevalence of list-adaptive compression technique, multiple levels of skip lists are seldom deployed.

Overall, skip pointers are an important requirement of an inverted index posting list implementation, and usable for many techniques that implement conjunctive query processing (e.g., see Section 2.2.1 below), or for dynamic pruning techniques (discussed later in Chapter 3).

2.2 Query Processing

When a query q is received, the role of the IR system is to select the docids to return to the user, based on the terms within the query. In a strict interpretation, the query would define exactly the terms that the retrieved documents must (or must not) contain. However, as elicited by Rijsbergen (1979), information retrieval differs from data retrieval (as might be performed using a database system) in that such an exact retrieval *"may sometimes be of interest but more generally we want to find those items which partially match the request and then select from those a few of the best matching ones"*. Hence, the query may be pre-processed (or automatically re-formulated) before retrieval commences, for instance, by removing stopwords, applying a stemming algorithm to identify additional similar words, or other automatic query expansion or query rewriting techniques.

Assuming that q has thus been pre-processed, in the remainder of this Section, we highlight the foundational query processing strategies that result in the docids being retrieved: for *Boolean retrieval*, when an exact answer is required (Section 2.2.1), and for *ranked retrieval*, when documents must be scored in relation to their similarity to q, to identify those to be returned to the user (Section 2.2.2). Note that techniques for rendering the results to the user, such as presenting the metadata for each document in the search

[9] The notion of the skip lists were introduced by Pugh (1990) and are probabilistic binary search tree data structures.

engine's result page, or the query-biased summarisation of the retrieved documents (which has been surveyed by Nenkova and McKeown (2011)) are outwith the scope of this survey. Next, we present and discuss the most recent comparisons of ranked retrieval strategies in terms of efficiency (Section 2.2.3). This is followed in Section 2.2.4 by an overview of complex queries, such as phrase and proximity queries, generated using a Web search engine's advanced query languages, and a survey of existing techniques to handle the processing of such queries. We conclude the section with a discussion on the multi-core query processing solutions (Section 2.2.5).

2.2.1 Boolean Retrieval

During its processing, a query expressed as a (multi-)set of terms, is processed against an inverted index to produce a list of documents that are returned to the user. In *boolean retrieval*, queries can be processed in *conjunctive* (AND) modes, in *disjunctive* (OR) mode, or a mix of the two modes.[10] In such a scenario, a conjunctive query processing algorithm must return a list of documents in which all the terms of the query appears at least once. Conversely, a disjunctive query processing algorithm must return all documents in which at least one query term appears.

Let us focus on conjunctive processing first. A simple and effective *binary intersection* algorithm for boolean conjunctive processing of two posting lists requires the parallel traversal of both posting lists, retaining the docids present in both posting lists. In its simplest implementation, the binary intersection algorithm scans the shorter posting list, and locates each element of the shorter posting list in the longer one. Portions of the long list can be skipped leveraging the (d) operator to avoid decompression and/or disk accesses to unwanted parts of the longer posting list.

With more than two posting lists, the binary intersection algorithm can be iteratively applied to compute the final results, as shown in Algorithm 2.1. The two shortest posting lists are intersected, then the resulting posting list is repeatedly intersected with the remaining posting lists in increasing order of length (lines 3–11). Since the posting lists are processed in order of increasing length, skipping over the longest posting list will be maximised, since, as the algorithm proceeds, fewer and fewer docids will fall into the intersected posting list (line 6).

Culpepper and Moffat (2010) presented a *holistic* intersection algorithm called max that performs similarly to the iterative binary merge with uncompressed sorted sequences. This algorithm, presented in Algorithm 2.2, has a memory access pattern with less spatial locality, but leverages the possibility to larger jumps in the lists. Each docid in the smallest posting list is tested against the docids of the remaining posting lists (line 7) and is retained

[10] Some search engines rewrite user queries into complex field-based boolean expressions, such as the complex operators discussed in Section 2.2.4 below.

Algorithm 2.1: The *binary merge* boolean conjunctive algorithm

```
   Input  : An array p of $n \geq 2$ posting lists, one per query term,
            sorted by increasing length
   Output : A posting list a of docids, sorted in increasing order
   BooleanAnd(p):
1      a ← p[0]
2      for i ← 1 to $n - 1$ do
3          tmp ← an empty posting list
4          current ← a.docid()
5          while current ≠ ⊥ do
6              p[i].next(current)
7              if p[i].docid() = current then
8                  tmp.insert(current)
9              a.next()
10             current ← a.docid()
11         a ← tmp
12     return a
```

if it is present in all the lists (lines 17–21). The algorithm maintains a `current` docid as it proceeds, stepping through the docids of the posting lists. When a docid greater than the `current` one is encountered (line 7), the shortest posting list iterator is moved to the next document whose docid is greater than or equal to this docid, and the next `current` docid is the larger between the shortest posting list new docid and the encountered docid (lines 9–14). The (d) operator is used to skip over whole portions of posting lists, pausing only at the `current` docid in each posting list (line 6).

Next, considering boolean disjunctive query processing, a basic algorithm is illustrated in Algorithm 2.3. In this case, no optimisations are possible. All posting lists must be traversed in parallel (lines 5–7), and every docid must be retained, taking into account that the same docid could appear in multiple posting lists. In order to traverse the posting lists, the algorithm maintains a state `current` as it proceeds (lines 2 and 8), where the smallest docid yet to be processed appearing in the posting lists is maintained by the MinimumDocid(p) procedure.

2.2.2 Ranked Retrieval

Web searchers often look only at the top few pages of results for a query (Silverstein *et al.*, 1999). Moreover, it is not feasible to return all matching documents to users, since the list of results could be potentially huge. As such, in ranked retrieval, the matching documents are ranked against the query according to some similarity function and just the K documents with the highest scores are returned to the users, using a generic ranking function as in Equation (2.1).

Algorithm 2.2: The *holistic* boolean conjunctive algorithm

```
Input  : An array p of $n$ posting lists, one per query term,
         sorted by increasing length
Output : A priority queue q of docids, sorted in increasing order
BooleanAnd(p):
    q ← a queue of docids, sorted in increasing order
    current ← p[0].docid()
    i ← 1
    while current ≠ ⊥ do
        for i < $n$ do
            p[i].next(current)
            if p[i].docid() > current then
                p[0].next(p[i].docid())
                if p[0].docid() > p[i].docid() then
                    current ← p[0].docid()
                    i ← 1
                else
                    current ← p[i].docid()
                    i ← 0
                break
            i ← i + 1
        if i = $n$ then
            q.push(current)
            p[0].next()
            current ← p[0].docid()
            i ← 1
    return q
```

Algorithm 2.3: The boolean disjunctive algorithm

```
Input  : An array p of $n$ posting lists, one per query term,
         sorted by increasing length
Output : A priority queue q of docids, sorted in increasing order
BooleanOr(p):
    q ← a priority queue of docids, sorted in increasing order
    current ← MinimumDocid(p)
    while current ≠ ⊥ do
        q.push(current)
        for i ← 0 to $n - 1$ do
            if p[i].docid() = current then
                p[i].next()
        current ← MinimumDocid(p)
    return q
```

The two classical query processing strategies to match documents to a query in ranked retrieval fall into two categories: *term-at-a-time* (TAAT) and *document-at-a-time* (DAAT) (Heaps, 1978; Buckley and Lewit, 1985; Turtle and Flood, 1995; Moffat and Zobel, 1996; Kaszkiel and Zobel, 1998). In the TAAT strategy (also known as *term-ordered* processing), the posting lists of the query terms are processed and scored sequentially to build up the result set. In the DAAT strategy (also known as *document-ordered* processing), the query term posting lists are processed in parallel, keeping them aligned by docid. The boolean processing algorithms seen in Section 2.2.1 are similar to DAAT, as they process all posting lists at once. TAAT versions of such algorithms are clearly possible.

When processing a query in conjunctive mode in ranked retrieval, no special differences arise with respect to the boolean retrieval. When the final list of results is computed by intersecting the posting lists, they are scored one by one, then a sorted list of the top K documents are returned. When a query is to be processed in disjunctive mode, the boolean retrieval algorithm in Algorithm 2.3 must take into account the management of the scores of the processed documents as well as the identification of the top K documents with the highest scores. We now illustrate the TAAT and DAAT query processing strategies in disjunctive mode, while in Section 2.2.3 we will summarise and compare their performances according to the existing literature.

The TAAT strategy is described in Algorithm 2.4. The posting lists are processed one by one (line 2) and, for a given document, its final score will be available once all posting lists have been processed. To temporarily store the *partial scores* of documents, a set of *accumulators* `A` is used (lines 1 and 5). Each accumulator contains the partial similarity score for a particular document computed thus far. The accumulators can be stored in a static array, indexed by docid or in a dynamic array, implemented via AVL trees, hashing and skip lists (Doszkocs, 1982; Harman and Candela, 1990; Moffat and Zobel, 1996; Cambazoglu and Aykanat, 2006). Once all documents appearing in the posting lists have been completely scored, the K documents with the highest scores are selected and returned (lines 8–10).

Early IR systems used the TAAT strategy in conjunction with the *cosine measure* as ranking function (Salton *et al.*, 1975). In such a ranking function, the term-document similarity $s_t(q, d)$ in Equation (2.1) is characterised by a *document normalisation weight*, and the ranking function can be factored as follows:

$$s_t(q, d) = W_t(q, d) \frac{1}{W(d)} \tag{2.3}$$

In such cases, the document normalisation weights were typically pre-calculated and stored at index construction time. To reduce the number of floating point operations, TAAT implementations with similar ranking functions sum up in the accumulators only the term-dependent contributions $W_t(d)$, postponing the document-only normalisation to a further final loop, traversing the final set of accumulators before the top K documents

Algorithm 2.4: The TAAT algorithm

```
    Input  : An array p of $n$ posting lists, one per query term
    Output : A priority queue q of (at most) the top $K$ ⟨docid, score⟩ pairs,
             in decreasing order of score
    TermAtATime(p):
 1      A ← an accumulators map from docids to scores,
              all entries initialised to 0
 2      for i ← 0 to $n - 1$ do
 3          current ← p[i].docid()
 4          while current $\neq \perp$ do
 5              A[current] ← A[current] + p[i].score()
 6              p[i].next()
 7              current ← p[i].docid()
 8      q ← a priority queue of (at most) $K$ ⟨docid, score⟩ pairs,
              sorted in decreasing order of score
 9      foreach ⟨docid, score⟩ in A do
10          q.push(⟨docid, score⟩)
11      return q
```

selection (between lines 7 and 8) (Moffat and Zobel, 1996).

Figure 2.5 illustrates an example of the TAAT strategy with 3 posting lists that returns the top 2 documents. Firstly, the shortest posting list p[0] is traversed, and the partial scores for docids 1 and 3 are computed and stored in the corresponding accumulator. Then, the second posting list p[1] is processed, updating the partial scores for docids 1 and 3, and creating a new accumulator for docis 2. Eventually the last posting list p[2] is processed, updating accumulators for docids 1 and 2, and creating a new accumulator for docid 4. To conclude, the priority queue q with the highest scores in decreasing order, i.e., docids 3 and 1, is computed and returned.

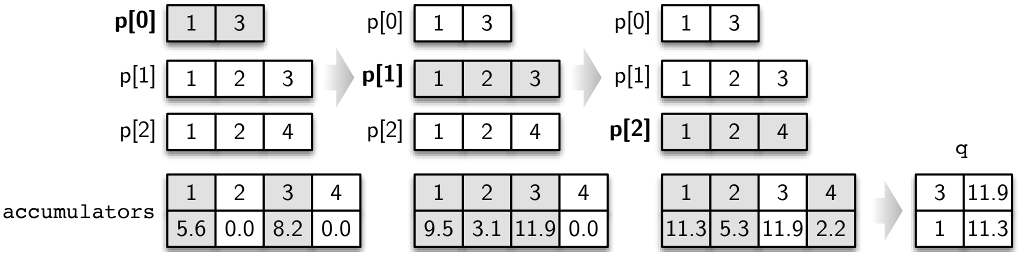


Figure 2.5: How the TAAT algorithm processes three posting lists

TAAT sequentially scans and processes a posting list at a time, which results in caching and prefetching benefits. However TAAT has some serious performance limitations when the document collection is huge, due to the memory required to store accumulators for

Algorithm 2.5: The DAAT algorithm

```
Input  : An array p of $n$ posting lists, one per query term
Output : A priority queue q of (at most) the top $K$ $\langle$docid, score$\rangle$ pairs,
         in decreasing order of score
DocumentAtATime(p):
1    q ← a priority queue of (at most) $K$ $\langle$docid, score$\rangle$ pairs,
         sorted in decreasing order of score
2    current ← MinimumDocid(p)
3    while current $\neq \perp$ do
4        score ← 0
5        next ← $+\infty$
6        for i ← 0 to $n - 1$ do
7            if p[i].docid() = current then
8                score ← score + p[i].score()
9                p[i].next()
10           if p[i].docid() < next then
11               next ← p[i].docid()
12       q.push($\langle$current, score$\rangle$)
13       current ← next
14   return q
```

each document in the collection. Moreover, the final top K documents selection could be a performance bottleneck, due to the scan of the final accumulators. Instead of sequentially scanning the posting lists, an alternative for disjunctive ranked retrieval consists in processing the posting lists in parallel, and fully scoring each document as soon as its postings are identified. This alternative lies at the core of the DAAT algorithm.

Algorithm 2.5 describes the DAAT algorithm. The smallest docid among the first docids of all posting lists is used to initialise the current docid (line 2). Then, all posting lists are traversed in parallel (lines 6–11), and the score for the current docid is computed by checking the iterator on each posting list (line 8). If a posting list's iterator is used to compute the score, it is then advanced, and the next docid to process is stored, leveraging the increasing docid sorting of the posting lists (line 9). After a docid is completely scored, it is immediately stored in the top K priority queue, sorted in decreasing order of score, and the algorithm proceeds to evaluate the next docid (lines 12–13).

An example of the DAAT strategy is presented in Figure 2.6, with 3 postings lists that returns the top 2 documents. Initially the priority queue q is empty, and the minimum docid is 1. Such docid is scored through the posting lists' iterators, hence the iterators are moved forward. The docid is inserted in the queue. The second docid to score is 2, and it makes its way to the queue as well. Then, as the processing proceeds, the docid 3 is processed, and it is inserted in the queue, removing the lowest scored docid in the queue, as it will not be returned as a result. Finally, the last docid is processed, but it is not inserted in the queue since its score is lower than the minimum score of the documents in the queue.

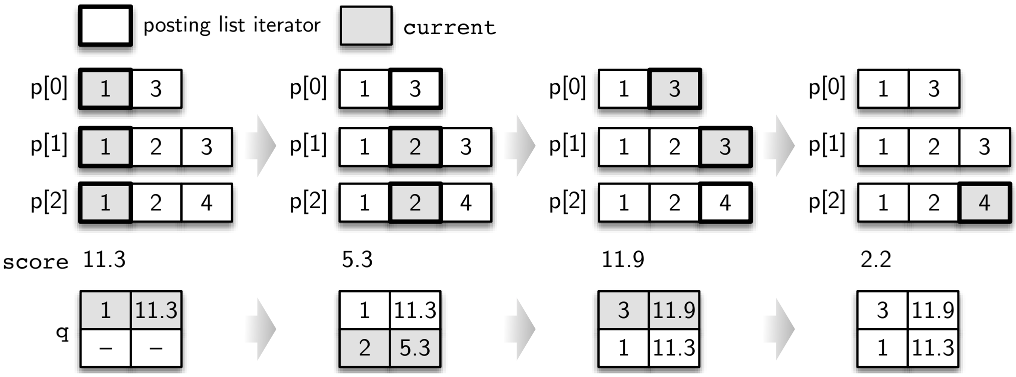


Figure 2.6: How the DAAT algorithm processes three posting lists

2.2.3 TAAT versus DAAT

The TAAT and DAAT strategies have been the cornerstone of query processing in IR systems since the 1970s. The plain implementations of these two strategies are not used anymore, since many optimisations have been proposed during the years (see Chapter 3), but several known systems in production today, from large-scale search engines such as Google and Yahoo!, to open source text indexing packages such as Lucene and Indri, use some optimised variations of these strategies (Fontoura *et al.*, 2011).

Noreault *et al.* (1977), Perry and Willett (1983), and Buckley and Lewit (1985) compared boolean versus ranked retrieval in early IR systems, and introduced the TAAT query processing strategy. Turtle and Flood (1995) are the first to argue that DAAT could beat TAAT in practical environments. Most of the research literature devoted to the DAAT and TAAT strategies is outdated, having compared the efficiency of both approaches together with their optimisations mainly on machines with limited memory and disk-resident indexes. Moreover, the typical index size in these works is limited to hundreds of megabytes. Fontoura *et al.* (2011) illustrated for the first time the performance results for TAAT and DAAT algorithms used in Yahoo!'s production platform for online advertisement. Given the sub-second latency requirements typical of commercial Web search engines, all experiments were conducted with memory-resident indexes. Their experiments were carried out on two test indexes: a small index of ∼200,000 documents (269 MB), and a larger index of more than 3 millions of documents (3.2 GB). The queries were divided in two sets: a query set containing ∼16,000 short queries (with mean query length of 4.26 terms), and a query set containing ∼11,000 long queries (with mean query length of 57.76 terms). The results they reported in the paper are summarised in Table 2.1, where the processing times, averaged on three runs, of the different strategies w.r.t. the indexes and query sets are reported (in microseconds). The naïve TAAT strategy always outperformed the naïve DAAT strategy, except for short queries in large indexes, where DAAT performs slightly better (∼5%) than TAAT.

Hence, TAAT is in general better than DAAT in terms of query processing time. However, the dynamic pruning optimisation techniques that will be discussed in Section 3.1 will show that the optimised DAAT strategies perform better than the optimised TAAT strategies.

Table 2.1: Latency results (in ms) for TAAT and DAAT on a small index (up) and a large index (down). Adapted from (Fontoura *et al.*, 2011).

	Small index		Large index	
	Short queries	Long queries	Short queries	Long queries
TAAT	**0.14**	**1.69**	3.78	**18.91**
DAAT	0.19	4.55	**3.58**	26.78

2.2.4 Complex Queries

Boolean conjunctive queries, as described in Section 2.2.1, are comparatively rare among the large query volumes serviced by IR systems, since users seldom use the advanced functionalities provided by the search engine's query language. However, many search engines provide an advanced query language supporting operators such as phrase or proximity queries. Moreover, it is common for search engines to *rewrite* the user's query into a richer low-level implementation, for instance to add phrasal or conjunctive operators when appropriate. In doing so, the search engine aims to improve the effectiveness of the query, and may also have the added benefit that the rewritten formulation of the query is more precise and can benefit the effectiveness of the search engine, and/or is more specific in terms of matching requirements, such that the efficiency of the query is improved.

Various open source search engines implement advanced complex query operators. Often, as in the case of the Indri/Galago search platform's query language Croft *et al.*, 2009, Ch.7,[11] these are not intended for use by end-users. Table 2.2, from (Macdonald *et al.*, 2017), shows examples of complex operators implementing synonym terms, phrasal terms and proximity terms. Examples of how these operators might be used to rewrite the query 'poker tournament' are shown in Table 2.3.

At the simplest level, query processing techniques that can answer queries using such complex operators can be implemented by surfacing each complex term in a query as a

[11] Also implemented by the Terrier IR platform (Macdonald *et al.*, 2018).

Table 2.2: Example of complex query operators.

Complex operator	Complex term	Description	Ephemeral posting list
#syn	#syn(car, cars)	Synonym set	Disjunctive/OR
#1	#1(new, york)	Phrase	Conjunctive/AND
#uwλ	#uw8(divx, codec)	Unordered window	Conjunctive/AND

Table 2.3: Example of rewrites for the query ‘poker tournament’ using complex operators.

Original query:	poker tournament
Stemming:	poker #syn(tournaments tournament)
Proximity:	poker tournament #1(poker tournament)$\wedge$0.1
	#uw8(poker tournament)$\wedge$0.1
Stemming and	poker #syn(tournaments tournament)
Proximity:	#1(poker #syn(tournaments tournament))$\wedge$0.1
	#uw8(poker #syn(tournaments tournament))$\wedge$0.1

posting list. For instance, the complex term ‘#1(new, york)’ can be surfaced as a single *ephemeral* posting list, which internally processes the posting lists for the constituent terms ‘new’ and ‘york’ in a conjunctive manner (see Figure 2.7), while determining in a DAAT fashion if the occurrences of the terms are adjacent. This can be achieved, for instance, by using Algorithm 2.1, and when the docids match (line 7), checking the position information stored within the postings of each constituent term to check for adjacency. For instance, in Figure 2.7, in docid 0, the term ‘new’ appears at positions 1 & 5, while ‘york’ appears at position 2 & 6. This means that the within-document frequency of such a pair of query terms in docid 0 is 2. While the terms appear in documents 3 & 5, they do not reoccur until document 6 (positions 4 & 5).

For phrasal and proximity operators with n terms, the calculation of the adjacencies can be formulated as a set intersection problem for n sorted sets of integers (each term’s position information will be sorted in ascending order). Asadi and Lin (2013) stated that this can be implemented using a *small adaptive* algorithm (Demaine *et al.*, 2000; Barbay *et al.*, 2006). When $n \geq 3$, the problem becomes harder, which Lu *et al.* (2015) addressed by using adaptations of a ‘plain-sweep’ algorithm originally proposed by Sadakane and Imai (1999).

Finally, it is of note that some large-scale search engines will prefer to record exact posting lists (rather than generate ephemeral postings lists) for some n-grams. For example Risvik *et al.* (2013), in their description of the Maguro part of the Bing search engine stack, described the indexing atoms (uni-grams or n-grams) that their engine will create posting

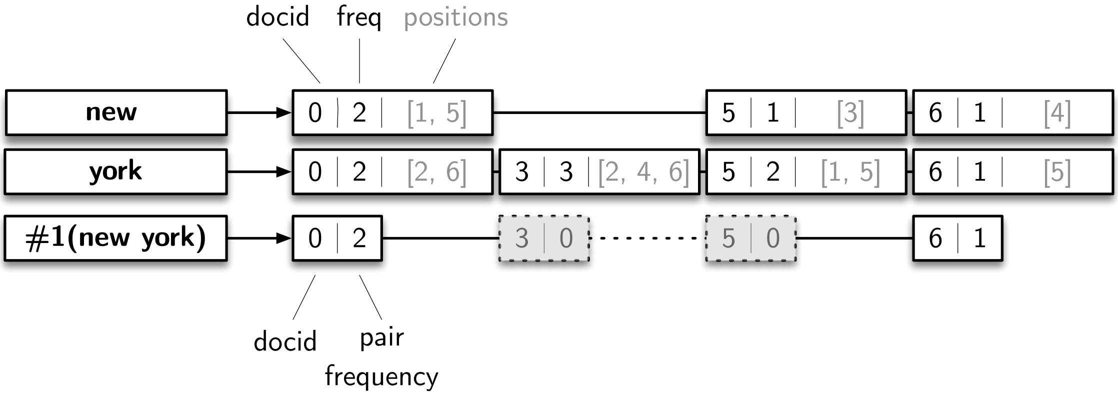


Figure 2.7: On-the-fly creation of an ephemeral postings list for '#1(new, york)', based on the positional posting lists for terms 'new' and 'york'.

lists for:

1. *All unigrams/words in the stream are atoms.*
2. *All n-grams (up to a given n) are considered as atoms.*
3. *A selection of all n-grams (beyond length n) are considered as atoms.*
4. *A selection of known n-grams of arbitrarily length are considered as atoms.*
5. *A selection of all tuples (w_i, w_j) are considered as atoms.*

The selection process for such n-grams and tuples uses n-grams identified a-priori from past query logs as well as from the document corpus itself Risvik *et al.* (2013).

2.2.5 Multi-core Query Processing

Typically, query processing is carried out sequentially, i.e., a single processor processes a single query at a time. With the advent of modern multi-core servers, query processing can be parallelised in several ways. The simplest way to leverage multi-core servers is to execute a query processing thread per core, and schedule each incoming query on the next available processing thread (Bonacic *et al.*, 2008; Frachtenberg, 2009; Tatikonda *et al.*, 2011). In this way, each core processes sequentially a single query (or a small batch of queries (Bonacic *et al.*, 2008)), and queries are dispatched to different cores using a simple first-come, first-served strategy. This parallelisation strategy incurs a minimal parallelisation overhead.[12] The throughput of the multi-core server increases almost linearly with the number of cores, up to ∼7.9× on a server with 8 cores (Bonacic *et al.*, 2008). However, this parallelisation strategy is unable to leverage extra processing capabilities from the available cores for processing a given query, resulting in no benefits in terms of query processing

[12] The inverted index is accessed in read-only mode.

times (Frachtenberg, 2009; Tatikonda *et al.*, 2011).

In order to leverage multiple cores to improve the response times of individual queries, a first solution consists in partitioning the document collection among processing threads. Frachtenberg (2009) proposed a *logical partitioning*, where all threads share the same index but each thread receives an equal-sized subset of documents to scan, together covering the entire inverted index. Tatikonda *et al.* (2011) proposed a *physical partitioning*, where the documents are divided into equally-sized partitions. An independent inverted index is built for every partition and assigned to its own processing thread. These partitioning schemes are unable to obtain linear increases in throughput: both schemes obtain an improvement of only $\sim$5.1× on a 8 cores server. On the other side, the mean query latency of both schemes leads to a speedup of 4.9× on 8 cores.

As an alternative to the *coarse-grained* parallelism among large document partitions exploited by logical and physical partitioning, Tatikonda *et al.* (2011) also proposed a *fine-grained* parallelism strategy among small-sized processing tasks. In this strategy a *producer task* receives a query, fetches the associated posting lists and generates multiple independent *intersection tasks*, as follows: Assuming all posting lists have uncompressed skip lists (see Section 2.1.2), an intersection task must compute the intersection among the postings between two consecutive skip pointers of the shortest posting lists and the remaining posting lists. The intersection tasks are collected into a shared *intersection task pool*, and the *consumer tasks* iteratively process them independently, generating intersection results that are added to a *scoring task pool*. The consumer tasks are also responsible for processing the scoring tasks and compute the final top results. The production of the intersection tasks is controlled by a capacity threshold, i.e., the number of intersection tasks in the pool at any time. When the number of tasks in the intersection task pool reaches the threshold, the consumer stops producing new intersection tasks and acts as a consumer. It starts producing new intersection tasks when the intersection task pool size falls below the capacity threshold. The fine-grained parallelisation obtains a throughput of up to $\sim$7.6× on a 8 cores server with a capacity threshold of 150 tasks, and a mean query latency speedup of 5.8× with a capacity threshold of 5 tasks.

A fixed number of cores, i.e., parallelism degree, is not suitable for all query loads. Parallelisation introduces computational overheads due to synchronisation and partitioning, negatively impacting both the query latency and the system throughput. Jeon *et al.* (2013) proposed an *adaptive parallelisation* strategy that selects the degree of parallelisation on a per query basis, depending on the number of available cores, the instantaneous system load and a request execution time profile. The request execution time profile $t()$ of a query associates the parallelism degree to an expected query processing time, i.e., $t(p)$ represents the expected query processing time when processed by p threads. Since the exact processing time of a query is not known in advance, the average processing time of past queries is used as an estimate. The adaptive parallelisation strategy selects, for a given query, the parallelism degree that increases the least the total processing time of all queries, i.e., that minimises the query latency of the queries currently processed by the server and the new query to process. The documents are sorted by decreasing static scores, and then they are logically partitioned into smaller subsets of documents called chunks. When a query arrives and its parallelism degree p is computed, p threads in parallel process the chunks, in decreasing order of static score. When a thread finds that the current top results are good enough and the later chunks are unlikely to produce better results, the thread early terminates the query processing, reducing the computational overhead. Their experiments, prototyped in Microsoft Bing and evaluated experimentally with production workloads, show a 2× speedup of mean and tail response latencies (e.g., the 95-th or 99-th percentile) for low and moderated workloads. Jeon *et al.* (2014) further extended the adaptive parallelisation approach by leveraging query efficiency predictors (see Section 4) in place of request execution time profiles. Their *predictive parallelisation* strategy parallelises only those queries that are predicted to be long-running (i.e., if the predicted execution time is greater than a given threshold) and runs the other queries sequentially. This strategy exhibits a 2× speedup for tail latencies, and an increase in system capacity of more than 50%. Kim *et al.* (2015) adopted a different efficiency prediction strategy for parallelisation. Each query is sequentially processed for a short time (e.g., 10 ms) and the parallelism degree is chosen after that initial processing, based on query efficiency predictors leveraging information collected during the short sequential processing. On an evaluation using a simulator, their *delayed-dynamic-selective parallelisation* strategy obtained comparable results to predictive parallelisation for mean latencies (up to 300 QPS) and tail latencies (up to 400 QPS). However, the new strategy is able to maintain the same latency performance for higher workloads, up to 600 QPS.

2.3 Summary

This chapter provided the necessary introduction to the foundational infrastructure inherent to every IR system. However, users are only interested in the top K ranked results identified in response to a user's query. In the next chapter, we provide an in-depth survey of optimisations that can be applied within an IR system to improve the efficiency. Indeed, many of these optimisations such as dynamic pruning – discussed in the following chapter– are made feasible by the top K nature of retrieval, i.e., by avoiding the indexing or scoring of documents that are unlikely to make the top K results.

3 Dynamic Pruning Query Processing

Since users are mostly interested in the top few pages of results for a query, the complete scoring of every document that contains at least one query term results in high latency. However, not all of these documents will make the top K retrieved set of documents that the user will see.

Two main general approaches have been exploited to increase the efficiency of query processing in IR systems in the top K ranked retrieval scenario: (1) avoid wasting time in processing portions of the inverted index containing documents that are unlikely to be relevant, (2) improve the efficiency of algorithms when processing portions of the inverted index containing relevant documents.

One of the main design solutions for dealing with these two approaches can be implemented at the query processing level, i.e., by modifying the behaviour of the retrieval algorithms to try to *prune* documents that will not be retrieved in the top K . In this chapter, we summarise the growing literature on *dynamic pruning* optimisation techniques that improve the query processing efficiency for both the TAAT and DAAT retrieval strategies.

In the following sections, we will introduce the main premises behind dynamic pruning (Section 3.1), which aims to dynamically skip documents stored in the inverted index that have a low chance to make the top K final results. We cover the TAAT and DAAT dynamic pruning optimisations respectively in Sections 3.2 & 3.3, including the popular WAND technique. Some of these techniques, as we will see, exploit new statistics computed on the inverted index. Section 3.4 discusses the most recent improvements to the optimisation techniques discussed in Section 3.3. These improvements leverage a measure of the contribution of portions of posting lists to the relevance of their documents for user queries, not by altering the inverted index, but instead by introducing a new component, the block max index. New improved query processing algorithms exploiting this new index component will conclude Section 3.4. Finally, Section 4 discusses techniques to *predict* how long a dynamic pruning strategy will take to execute a query.

3.1 Introduction to Dynamic Pruning

While static pruning strategies (discussed in Chapter 5) alter the index structure at index construction time, dynamic pruning aims to alter query processing in such a way that potentially non-relevant documents, for a given query, can be efficiently ignored. Although different dynamic pruning strategies have been proposed for TAAT and DAAT strategies through the years, all of these optimisations rely on a common observation, namely that as soon as it can be determined that a document will never be able to enter in the final top K results, we can ignore it during processing, or stop its current processing. This observation, sometimes referred to as the *early termination condition*, the *stopping condition* or the *pruning condition*, can be stated more clearly by introducing the following definitions:

- *early termination*: during the processing of a query, a document evaluation is early terminated if all or some of its postings, as defined by the terms of the query, are not fetched from the inverted index or not scored by the ranking function.
- *term upper bounds*: for each term t in the vocabulary, we compute a term upper bound (also known as *max score*) $\sigma_t(q)$ such that, for all documents d in the posting list of term t,
$$\sigma_t(q) \geq s_t(q,d) \tag{3.1}$$
The term upper bounds can be computed offline by taking the maximum score value of the highest scoring document for each term in the vocabulary and storing the observed score in the vocabulary. Alternatively, for some similarity measures, term upper bounds can be quickly estimated at runtime (Macdonald *et al.*, 2011).
- *document upper bounds*: given a similarity function such as in Equation (2.1), for a query q and a document d, we can compute a document upper bound $\sigma_d(q)$ based on the terms occurring in the document, by summing up the n term upper bounds:
$$\sigma_d(q) = \sum_{t \in q} \sigma_t(q) \tag{3.2}$$
During query processing, we compute the final query-document score by sequentially computing the query-term scores. As soon as the postings of some terms in $\hat{q} \subseteq q$ have been scored, we can compute a lower upper bound:
$$\sigma_d(q) = \sigma_d(q, \hat{q}) = \sum_{t \in \hat{q}} s_t(\hat{q}, d) + \sum_{t \in q \setminus \hat{q}} \sigma_t(q) \tag{3.3}$$
- *thresholds*: during query processing, the top K full or partial scores computed so far, together with the corresponding docids, are organised in a separate data structure, and the smallest value of these (partial) scores is called threshold θ. If there are not at least K scores, the threshold value is assumed to be 0. This data structure can be implemented as a priority queue `queue` with capacity K (also known as *max-heap*), supporting the following operations:
  - `queue.push(⟨docid, score⟩)`, adding the ⟨`docid`, `score`⟩ pair to the queue if the score is greater than the current threshold, and evicting, if the queue is full, the least scoring docid.
  - `queue.min()`, returning the value of the current threshold θ, or 0 if the queue is not full.
  - `queue.pop()`, removing the top scoring ⟨`docid`, `score`⟩ pair from the queue and returning it.

  The threshold has the fundamental property of *non-negative monotonicity*.[1] Indeed, during query processing, its value always increases: as new documents are added to the

[1] We assume $s_t(q,d) \geq 0$ for all queries and documents.

queue, their scores cannot be smaller than the current threshold θ. If the document score is equal to θ, typically the document, which has a greater docid, is not added to the queue.

At this point we can formulate the *pruning condition*: for a query q and a document d, if the document upper bound $\sigma_d(q)$, computed by using partial scores, if any, and term upper bounds, is less than or equal to the current threshold θ, the document processing can be early terminated, i.e., if the condition

$$\sigma_d(q) \leq \theta \tag{3.4}$$

evaluates to false.

In essence, the focus of dynamic pruning is to bring the benefits of skipping from conjunctive query processing to disjunctive query processing. In other words, all dynamic pruning strategies aim to process queries conjunctively when possible, and disjunctively otherwise.Dynamic pruning strategies work well when queries are composed of highly discriminative (i.e., rare) terms as well as poorly discriminative (i.e., common) terms. For example, when processing the query "*mississippi river cruise*", the term *mississippi* will typically be rarer in the collection and hence it will make a higher score contribution, while the other terms *river* and *cruise* will make a smaller contribution, since both terms are quite common. When processing disjunctively such a query, after a few documents are scored, it is clear that any new document, to be returned in the final top K results, must necessarily include the term *mississippi*, while the other two terms may not be present in the document. Hence, while all documents in the *mississippi* posting list must be processed, we can skip through the posting lists of the other two terms, focusing only on the documents containing *mississippi*. Posting lists are read in blocks, so it pays, in term of efficiency, to skip over large chunks of postings in the longer posting lists.

Figure 3.1 illustrates an early termination with 4 posting lists. In the example, the current theshold value θ is 6.0, resulting from documents in the current top results. While scoring docid 11, the document upper bound is initially set to 10.1, the sum of the 4 term upper bounds. While postings are processed, the document upper bound is adjusted with the actual scores computed for each term. Hence, it decreases to 8.5 after the posting from the first posting list is processed. Since $8.5 > 6.0$, we must continue to process the other posting lists. After the posting from the second list is processed, the document upper bound becomes 4.6. At this point we are sure that document 11 will never obtain a final score greater than 4.6 and since it is less than the current threshold, the postings of the last two lists can be ignored, and the processing of docid 11 can be terminated, proceeding to the next docid.

This pruning condition and how it is used can have further consequences on the particular query processing strategy adopted. Some optimisations may lead to degraded effectiveness. According to the terminology introduced by Turtle and Flood (1995) and

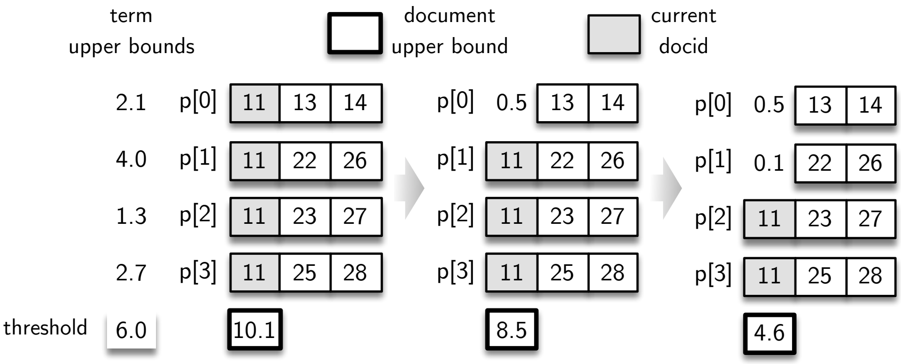


Figure 3.1: An example of early termination.

Strohman (2007), the optimisations can be classified into four classes, depending on their effectiveness guarantees:

safe/unoptimised: the optimisation guarantees that all documents, not just the top K, are ranked correctly, i.e., documents appear in the same order and with the same score as they would appear in the ranking produced by a unoptimised strategy. The computation of any effectiveness measure would be unaffected.

safe up to K/rank safe: the optimisation guarantees that the top K documents produced are ranked correctly, but the document scores are not guaranteed to coincide with the scores produced by an unoptimised strategy. These are the most interesting dynamic pruning optimisations, since they do not negatively impact the effectiveness of the documents returned to the users while introducing efficiency gains. Indeed, relevance evaluation metrics are typically computed over the top $K = 10, 20$ documents, such as MAP@10 or NDCG@20.[2]

unordered safe up to K/set safe: the documents returned by this optimisation coincide with the top K documents computed by a full strategy, but their ranking can be different.[3]

approximate/unsafe: no provable guarantees on the correctness of any portion of the ranking produced by these optimisations can be given, but most unsafe optimisations proposed thus far in the literature produce results with very limited effectiveness losses.

In the following, we will illustrate the main dynamic pruning optimisations separately

[2] If the optimisation guarantees that the top K documents produced are ranked correctly with the same score as they would appear in the ranking produced by an unoptimised strategy, it is also called *score safe*.

[3] Optimisations with such a guarantee can be considered as good as safe up to K when used in a learning cascade, since the actual ordering of the top K results is not important.

for the TAAT and DAAT strategies, with a particular focus on safe up to K strategies.

3.2 TAAT Optimisations

In the TAAT strategy, all postings of the posting lists of any term appearing in the query are processed (see Section 2.2.2). This means that, for any document appearing in any posting list, an accumulator must be created (and updated). Beside requiring time to process the postings lists, also the memory space required during query processing can be a problem, in particular if memory is a scarce resource or takes time to allocate.

All TAAT dynamic pruning optimisations split the query processing into two distinct phases. During the first phase, the normal TAAT algorithm is executed, one term at a time, in increasing order of document frequency. This order takes into account the term importance: shorter posting lists will get higher IDF and higher similarity scores (and consequently higher term upper bounds). This phase is similar to the processing of a disjunctive query and, as such, it is also called the *OR mode* phase. New accumulators are created and updated, until a certain pruning condition is met, and then the second phase starts. Then, no new accumulators are created, and a different algorithm is executed, on the remaining terms and/or on the accumulators created during the first phase. According to the terminology introduced in (Moffat and Zobel, 1996; Anh and Moffat, 1998), the second phase algorithms can be classified according to the following scheme:

Quit. The processing of postings completely stops at the end of the first phase. No new accumulators are created and no postings from the remaining terms' posting lists are processed. Some of the existing accumulators will contain only partial scores, because some score contributions could have been computed during the second phase.

Continue. The creation of new accumulators stops at the end of the first phase. The remaining terms' posting lists will be processed, but just to update the score of the already scored documents with new postings. This phase is similar to the processing of a conjunctive query, since we will look up for specific docids (those for which an accumulator has been created during the first phase) in the remaining posting lists, hence it is also called the *AND mode* phase.

Decrease. The processing of postings in the second phase proceeds as in Continue, but the number of accumulators is decreased as the remaining terms are processed. Those terms will have small score contributions, with few chances to alter the current top K document ranking. More importantly, the memory occupancy can be reduced as soon as we realise that an existing accumulator can be dropped since it will never enter in the final top K documents, with a corresponding benefit in terms of response times.

In the following, we will illustrate the different second phase algorithms and optimisations.

Quit and Continue

Smeaton and Rijsbergen (1981) proposed the first Quit optimisation,[4] further refined by Perry and Willett (1983), that we will call EarlyQuit. They proposed to store separately the top K accumulators encountered so far. The minimum value in this data structure is the current threshold θ. Partial scores of documents encountered during processing are stored in the accumulators. The posting lists are processed in increasing document frequency order. By doing this, early termination may omit the longer lists from scoring, thus increasing the efficiency of the algorithm. This sorting of the posting lists is exploited by all optimisations we will discuss. After looking at every document in a given term's posting list, a document upper bound among those documents not yet encountered is computed. This is accomplished by calculating, at runtime, the term upper bound for those terms $q \setminus \hat{q}$ that have not been processed thus far, and summing them together. If the current threshold is greater than this document upper bound, i.e.,

$$\theta > \sum_{t \in q \setminus \hat{q}} \sigma_t(q) \tag{3.5}$$

the TAAT algorithm is concluded, and the top K documents are returned, ranked by their partial scores. Hence, this optimisation is unordered safe up to K. Note that in this algorithm, which we will denote EarlyQuit, the pruning condition is tested every time a posting list is fully processed. The reported efficiency benefits are not appropriate for current collections, containing several orders of magnitude more documents.

A similar idea is further explored by Buckley and Lewit (1985) and Lucarella (1988). They proposed another Quit optimisation for the TAAT strategy that keeps track of the top $K+1$ partial scores. Every time a posting list is fully processed, the partial score θ of the K-th current top document is compared with the sum of the partial scores $\mathtt{A}[K+1]$ of the $(K+1)$-th current top document and the term upper bounds of the terms $q \setminus \hat{q}$ yet to process. If it is impossible that the top $(K+1)$-th document can beat the partial score of the top K document by using the max scores of the remaining query terms, i.e.,

$$\theta > \mathtt{A}[K+1] + \sum_{t \in q \setminus \hat{q}} \sigma_t(q) \tag{3.6}$$

then the processing can be stopped and the top K documents are returned, ranked by their partial scores. In this case too, the pruning condition is tested after the full evaluation of each posting list. Moreover, the optimisation is unordered safe up to K. This optimisation

[4] Actually, it is not clear from (Smeaton and Rijsbergen, 1981) if partial accumulators are fully scored or not, but we are inclined towards the Quit scheme.

can have problems if the K-th and $K + 1$-th documents have an identical (or very similar) score(s), since in that case it is impossible to early terminate the processing of the remaining posting lists. Turtle and Flood (1995) compared this optimisation w.r.t. TAAT on a small disk-resident index and queries with 9.1 terms on average, showing a 11.7% reduction in terms of processed postings ($K = 20$). Fontoura *et al.* (2011) provided a comparison between this optimisation and the normal TAAT strategy for memory-resident indexes. Although the optimisation was able to skip a few score computations, no benefits emerged w.r.t. TAAT for small indexes, and just around a $1.04\times - 1.07\times$ speedup on average latency was observed with large indexes ($K = 30$).

Instead of completely stopping the scoring process when the pruning condition is satisfied, Harman and Candela (1990) proposed to only process all postings of the most important terms of a query. In particular, they completely skipped terms whose IDF is less than a fraction of the maximum IDF among all terms of the query. This optimisation, denoted with MaxIDF, can be considered a Continue strategy, since the document scoring is not early terminated. However, this approach is approximate, since the posting lists of the ignored terms could change the score of the final top K documents. If the score difference between the last K-th top document and the $K + 1$ document is small, even a small score contribution from a low IDF term could change their relative ranking.

Decrease

Moffat and Zobel (1994) and Moffat and Zobel (1996) were the first to propose a systematic approach to leverage skipping to reduce the number of accumulators and to switch from a first OR-like query processing mode to a second AND-like mode. The core idea they proposed is to restrict with an a priori, query-independent bound L, the maximum number of accumulators that can be created during query processing. Once L accumulators have been created, two possible processing alternatives are discussed. In the Quit strategy, the processing of inverted lists is short-circuited, with great benefits on the processing efficiency but with a possibly poor effectiveness performance, since this strategy is approximate. Alternatively, in the Continue strategy, the processing of inverted lists continues, but no new accumulators are allowed. The documents corresponding to the L accumulators will be fully scored, but cannot be guaranteed to contain the top K documents of a safe ranking since the limit on the number of accumulators is fixed a priori. Nevertheless, it is reasonable to expect a very limited impact on the effectiveness for large values of L (Kaszkiel *et al.*, 1999).

Lester *et al.* (2005) noted that the pruning condition of both Quit and Continue completely rejects whole posting lists rather than separate documents within the lists, causing a *bursting* effect on the candidate set of accumulators, since the target number of accumulators L could be dramatically exceeded in some cases.[5] While the strategies checking the pruning condition at the end of the posting lists have been renamed Quit-Full and Continue-Full, two alternative strategies, namely Quit-Part and Continue-Part, check the number of accumulators' condition after every posting. However, new experiments conducted in a Web search scenario, i.e., queries with a small number of terms, not dozens like in (Moffat and Zobel, 1996), with corpora of Web documents and generic users, showed that the Continue-Full strategy did not perform well in terms of actual number of accumulators created, while the Continue-Part strategy did not perform very well in terms of effectiveness.

The first observation that the accumulators could be decreased as the later terms in the query are processed appeared in (Turtle and Flood, 1995). The authors proposed an approach called TAAT MaxScore similar to the one discussed by Smeaton and Rijsbergen (1981).[6] During the first phase, a threshold of the top K accumulators computed thus far is mantained. After processing each term, the threshold is checked against the sum of the upper bounds of the remaining terms. If the current threshold is greater than the sum of upper bounds, no documents that do not already have an existing accumulator can be in the top K final results. When this happens, instead of terminating the query processing, the remaining posting lists are processed with a Continue strategy. In this second phase, we only need to score postings of documents seen during the first phase, hence the accumulators must be stored in a sorted list. Moreover, the TAAT MaxScore strategy can also reduce the number of accumulators during the first phase. As soon as we can state that the current partial score $\mathtt{A}[d]$ of a document plus the upper bounds of the remaining terms cannot be greater than the current threshold, we can safely remove the document's accumulator from the candidate set, i.e.,

$$\theta > \mathtt{A}[d] + \sum_{t \in q \setminus \hat{q}} \sigma_t(q) \tag{3.7}$$

When this condition holds, document d will never be able to enter the final top K results, even if the current threshold is not the final one.

Turtle and Flood (1995) compared this optimisation w.r.t. TAAT, showing a 65.2% reduction in processed postings when $K = 20$. This benefit decreases to 60.8% when $K = 100$ and further to 44.4% when $K = 1,000$.

Anh and Moffat (1998) discussed another accumulator Purging strategy: at the end of the first phase, when we have L accumulators, the number of accumulators is reduced to a second fixed number L_0, and is periodically halved during the second phase. The halving in the second phase is designed to reach exactly K accumulators after the last query term is processed. As the authors note, this strategy is negatively impacted by

[5] Suppose that L is set to 1,000 accumulators for a two terms query. The first posting list contains 100 postings, while the second list contains 100,000 postings. During processing, 100,100 accumulators will be created.

[6] Not to be confused with the (DAAT) MaxScore strategy introduced in Section 3.3.

the cost of performing the purging. To make the process fast, they proposed to store the accumulators in an unordered array, but this solution requires a completely random access mechanism to posting lists, instead of a sequential access with skipping. This approach is only approximate, since there are no guarantees that the purging of accumulators will not remove some (unlikely) top K documents.

A further improvement in the dynamic management of accumulators has been proposed by Lester *et al.* (2005). They introduced an *adaptive pruning* strategy to control the memory usage in the TAAT strategy, denoted with Adaptive. At the beginning of the processing of any posting list (in decreasing order of document frequency), a *threshold value* v is computed. This threshold, depending on the actual scoring function, is directly dependent on a corresponding term-document frequency *hurdle* h. A posting's partial score greater than v must have a value of $f_{d,t}$ greater than h. New accumulators are created only if their partial scores are greater than v, while existing accumulators with partial scores lower than v are removed from the candidate set. The term-document frequency hurdle h must be adaptively estimated at runtime, depending on the size of the candidate set, the number of "good" postings accumulated thus far in the current posting list and the expected number of "good" postings in the remaining postings of the list. The experiments in a Web search scenario showed that, even if approximate only, the proposed adative pruning TAAT strategy gives very good effectiveness results by using a number of accumulators equal to 0.4% of the size of the collection. A further refinement to this algorithm, AdaptiveSkips, that leverages skipping, has been proposed by Jonassen and Bratsberg (2011).The authors reported a 1.33× speedup versus the original adaptive pruning strategy.

Later, Fontoura *et al.* (2011) introduced a modification to the TAAT MaxScore optimisation by noting that, while the TAAT MaxScore strategy was originally designed to skip over portions of disk-resident posting lists during the second phase, its benefits could be definitely less marked in memory-resident indexes. The main disadvantage of the original TAAT MaxScore is the overhead to update the candidate set during the first phase and to sort it before the second phase starts, in order to minimise the number of skips. Hence the authors proposed a *memory-resident* TAAT MaxScore optimisation (TAAT mMaxScore), where the candidate set is not sorted before the second phase and hence the remaining posting lists are scanned sequentially and merged during the second phase, with no skipping taking place. Moreover, they avoided pruning accumulators during the first phase. According to their experiments (summarised in Section 2.2.3), this optimisation performs always better than the Buckley and Lewit (1985) optimisation and the Turtle and Flood (1995) original TAAT MaxScore, resulting in a 1.09×−1.22× mean response time speedup for small indexes and a 1.50×−1.60× mean response time speedup for large indexes ($K = 30$) w.r.t. the normal TAAT strategy.

Table 3.1 summarises the TAAT optimisations discussed, together with their associated second phase processing scheme and their effectiveness guarantees. Even if modern Web

search systems do not use TAAT strategies anymore, the
xspace optimisation can be considered to be the best TAAT strategy with the current document collections.

Table 3.1: TAAT optimisations summary.

Name	Reference	2^{nd} phase	Effectiveness
EarlyQuit	(Smeaton and Rijsbergen, 1981) (Perry and Willett, 1983) (Buckley and Lewit, 1985) (Lucarella, 1988)	Quit	unordered safe up to K
MaxIDF	(Harman and Candela, 1990)	Continue	approximate
Quit-Full	(Moffat and Zobel, 1994; Moffat and Zobel, 1996)	Quit	approximate
Continue-Full	(Moffat and Zobel, 1994; Moffat and Zobel, 1996)	Continue	approximate
TAAT MaxScore	(Turtle and Flood, 1995)	Decrease	safe up to K
Purging	(Anh and Moffat, 1998)	Decrease	approximate
Adaptive	(Lester *et al.*, 2005)	Decrease	approximate
AdaptiveSkips	(Jonassen and Bratsberg, 2011)	Decrease	approximate
TAAT mMaxScore	(Fontoura *et al.*, 2011)	Continue	safe up to K

3.3 DAAT Optimisations

The DAAT strategy computes the full document scores in a single step, eliminating the need for a data structure to store the partial scores of documents as in the TAAT strategies. Moreover, this allows us to store at any time the top K documents seen thus far, and to compute dynamically the threshold θ on the full scores. All DAAT optimisations rely on the dynamic update and growth of this threshold, together with term upper bounds, to markedly improve the performance of the base DAAT algorithm. They also have a smaller memory footprint than the TAAT strategies, since at runtime they store $O(k)$ document scores instead of $O(N)$ accumulators. Even if they do not exploit the spatial locality of the posting lists and they results in high branch mispredictions due to the frequent comparisons, the DAAT optimisation strategies are known to be successfully adopted by several commercial Web search engines.

The DAAT optimisations can be broadly classified into four main classes, as follows:

Top Docs: a certain number of documents for each term are stored separately as the "most influential" documents for the given term, and used in a pre-processing phase to improve the efficiency of the subsequent DAAT processing.

Max Scores: a term upper bound is stored in the vocabulary for each posting list, and it is used at runtime to take decisions on the early termination and/or skipping of certain postings/documents.

Block Max: every posting list is divided into blocks. Each block is considered as a self-contained posting list, and assigned a max score. Then, during processing, the blocks are traversed first, and if their max scores indicate some potential top K documents, they are processed posting by posting, otherwise they are completely skipped.

In the following, we will illustrate different *top documents* optimisations, and the two most important *max scores* optimisations, namely MaxScore and WAND, while the discussion of the *block-based max scores* algorithms follows in Section 3.4. To conclude, we also provide some efficiency measures reported in the literature about the DAAT optimisations.

Top Documents

Brown (1995) proposed one of the early DAAT optimisations for disk-based inverted indexes. In order to minimise the number of disk reads during query processing, he proposed to focus processing on documents with high score contributions. For any given term t in the vocabulary, a *top candidates* list (also known as a *champions list*) is stored separately as a new posting list. The postings in that list are the top N_t results returned to the query t, by using the same scoring function that will be used at query time. The typical value of N_t is $1,000$, such that every term in the vocabulary ends up with an additional posting lists of $1,000$ postings, corresponding to the top $1,000$ highest score contributions in the original posting list. Then, the normal DAAT processing for any query is applied to the top candidate posting lists only, reducing markedly the number of postings scored, but with a potential negative impact on the effectiveness, since this optimisation is unsafe.

Strohman *et al.* (2005) proposed a safe up to K version of the top documents optimisation by merging it with a subsequent phase based on MaxScore. Therefore, we will present Strohman et al.'s strategy after we introduce MaxScore itself.

Fontoura *et al.* (2011) introduced a *family* of optimisations based on *top terms* instead of top documents. Given a query, it is split into two sets of terms, depending on their document frequency. Given a postings threshold T, all terms with higher IDF values, i.e., shorter posting lists ($f_t \leq T$), are processed in a first phase, while all other terms, those with lower impact, i.e., longer posting lists, are processed in a second phase. During the first phase, all documents in the short posting lists are processed in an OR-mode, using a TAAT or DAAT strategy, with no optimisations. The candidate set produced is viewed as another posting list in the second phase. During the second phase, the long posting lists and this candidate list are processessed using a DAAT optimisation such as MaxScore or WAND. Moreover, the K-th partial score of the candidate set is used to initialise the threshold of the max-heap priority queue used in the second phase. Kane and Tompa (2018) investigated a similar approach based on *split-lists* for WAND. During indexing time, they proposed to split each list into two parts, where the first part contains the highest scored documents and the second part contains the remaining documents. Their experiments reported a 1.7× speedup for WAND.

MaxScore

The MaxScore optimisation, introduced by Turtle and Flood (1995), is a safe up to K strategy aiming to boost the efficiency of the DAAT algorithm through the following observation. At some point during query processing, we can expect that the threshold will be large enough to prune documents appearing only in the posting list with the smallest term upper bound contribution. When this happens, the algorithm can safely skip over documents appearing only in that posting list, and can consider as top K candidate documents only those appearing in the remaining posting lists. Once a new candidate document must be fully scored, that posting list can be traversed in an *AND mode* to look for the candidate docid only. This observation can be applied to the remaining posting lists as the query processing proceeds and the threshold increases.

A possible implementation is reported in Algorithm 3.1, based on the description by Fontoura *et al.* (2011).[7] The algorithm takes as input two arrays of size n: the posting lists `p` to be processed and the corresponding term upper bounds `σ`. Both arrays are sorted in increasing order of max score. At runtime, the posting lists are kept separated into two sub-lists by a `pivot` index, running from 0 to $n-1$. The posting lists indexed from the `pivot` up to n form the *essential lists*, while the remaining posting lists, if any, are the *non-essential lists*. At any time during query processing, no document can be returned as a top K result if it only appears in the non-essential lists, i.e., at least one of the terms corresponding to the essential lists must occur in any top K document.

To update the `pivot`, we compute n document upper bounds `ub` (line 5). The entry `ub[0]` contains the document upper bound for documents appearing just in `p[0]`, the entry `ub[1]` contains the upper bound for documents appearing only in `p[0]` and `p[1]`, and so on. While there is at least an essential list and there are documents to process (line 9), the MaxScore algorithm first processes the essential lists selecting the candidate docid as in DAAT, while storing the next docid to process (lines 12–17). Then, it proceeds by processing the non-essential lists by skipping to the candidate docid (lines 18-23). As soon as the pruning condition holds (line 19), we are sure that the candidate docid cannot be in the final top K documents, and the remaining posting lists can be skipped completely. If all the non-essential posting lists are processed, we check if the final score is high enough to enter

[7] The original description of MaxScore (Turtle and Flood, 1995) does not include all the details of its implementation, and different descriptions have been later proposed (Strohman *et al.*, 2005; Lacour *et al.*, 2008; Jonassen and Bratsberg, 2011; Fontoura *et al.*, 2011).

in the current top K documents (line 19). If this happens, the current threshold could be updated (line 25), and the current pivot could change as well (lines 26–27).

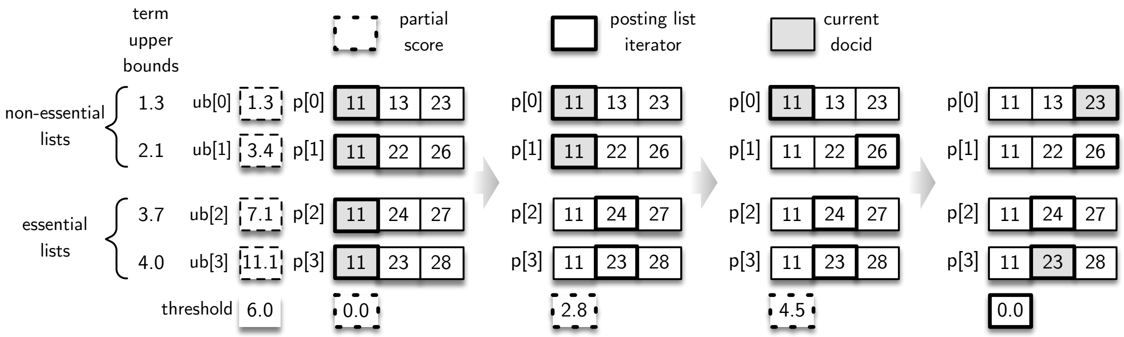


Figure 3.2: How the MaxScore algorithm processes four posting lists.

Figure 3.2 illustrates a MaxScore early termination with 4 posting lists. In the example, the current threshold value θ is 6.0, resulting from documents in the current top results. The posting lists are sorted by increasing term upper bound, and we have used the term upper bounds to compute the values of the array ub. Since ub[1] $= 3.4$ and ub[2] $= 7.1$, the pivot is set to 2, p[0] and p[1] are the non-essential lists, while p[2] and p[3] are the essential lists. We are processing the document with docid 11. Firstly, we process the essential lists and compute the partial score of the document (e.g., 2.8), keeping track of the next docid to process while advancing the posting list iterator of these lists by one step (e.g., 23). Since, given the partial score computed so far, it is possible that docid 11 could exceed the current threshold (e.g., $2.8 + \texttt{ub[1]} = 2.8 + 3.4 = 6.2 > 6.0 = \theta$), we start processing the non-essential lists for that document. After we process p[1], skipping directly to the next docid greater than or equal to the next docid to process (e.g., 23), we get an updated partial score of 4.5. Now, since ub[0]$= 1.3$, we can safely ignore the first posting list (e.g., $4.5 + 1.3 = 5.8 < 6.0$), and skip directly to docid 23 using the p.next(d) operator. As discussed in Section 2.1.2, skipping allows us to avoid reading and decompressing portions of the posting lists, with benefits in terms of efficiency proportional to the size of the skipped chunks. Note that in the example of Figure 3.2, we do not include the priority queue and pivot updates for simplicity.

Strohman *et al.* (2005) proposed a further optimisation of the MaxScore strategy, leveraging the top documents approach of Buckley and Lewit (1985). For every term t of the vocabulary with more than 1,000 postings, they extracted a smaller posting list, composed of the top scoring documents for the term t. The number of pruned documents for each term can be selected in different ways: a *fixed* number of documents from all posting lists, a constant *fraction* of the documents in each postings list or all the documents with a term-document *frequency* greater than a given per-term threshold. Nevertheless, a *top doc*

Algorithm 3.1: The MaxScore algorithm

```
Input  : An array p of $n$ posting lists, one per query term,
         sorted in increasing order of max score contribution
         An array σ of $n$ max score contributions, one per query term,
         sorted in increasing order
Output : A priority queue q of (at most) the top $K$ ⟨docid, score⟩ pairs,
         in decreasing order of score
MaxScore(p,σ):
    q ← a priority queue of (at most) $K$ ⟨docid, score⟩ pairs,
          sorted in decreasing order of score
    ub ← an array of $n$ document upper bounds, one per posting list,
          all entries initialised to 0
    ub[0] ← σ[0]
    for i ← 1 to $n - 1$ do
        ub[i] ← ub[i − 1] + σ[i]
    θ ← 0
    pivot ← 0
    current ← MinimumDocid(p)
    while pivot $<$ $n$ and current $\neq \perp$ do
        score ← 0
        next ← $+\infty$
        for i ← pivot to $n - 1$ do                              // Essential lists
            if p[i].docid() = current then
                score ← score + p[i].score()
                p[i].next()
            if p[i].docid() < next then
                next ← p[i].docid()
        for i ← pivot − 1 to 0 do                                // Non-essential lists
            if score + ub[i] $\leq$ θ then
                break
            p[i].next(current)
            if p[i].docid() = current then
                score ← score + p[i].score()
        if q.push(⟨current, score⟩) then                         // List pivot update
            θ ← q.min()
            while pivot $<$ $n$ and ub[pivot] $\leq$ θ do
                pivot ← pivot + 1
        current ← next
    return q
```

index is produced, including all the pruned posting lists. During query processing all of the documents in this index are processed, and stored with their scores in a max-heap priority queue. The priority queue will contain the union of the top documents and a partial score for each one, sorted by decreasing partial score. These partial scores are guaranteed to be less than or equal to the final score of the corresponding document (some contributions could still come from the remaining postings). Moreover, the partial score of the K-th document is used to initialise a threshold value for the subsequent phase, since any other document not already processed within the top documents must beat this threshold. Indeed, during the processing of the top documents, we have already scored the top scoring postings for the involved terms, and hence we can safely assume the *min score* of each term's top document lists to be the upper bound of the score of the remaining postings of that term. Hence, during the second phase, the top documents are completely scored, while the remaining documents are processed by MaxScore, with the initial threshold value and the modified term upper bounds. Like the MaxScore optimisation, this strategy, called Term Bounded MaxScore is safe up to K. In their experiments, the authors reported that the best top doc selection strategy is *frequency*, even if the disk space occupancy of the top index is difficult to predict. In that case, their Term Bounded MaxScore implementation obtained a 23% speedup with respect to MaxScore and a 61% speedup versus DAAT.

WAND

The WAND optimisation, introduced by Broder *et al.* (2003), stems from the definition of a new boolean operator, *Weak AND* or *Weighted AND*. The WAND operator takes as input a list of n boolean variables $X_0, \dots, X_{n-1}$, a list of n associated weights $w_0, \dots, w_{n-1}$ and a threshold θ. By definition, WAND $(X_0, w_0, \dots, X_{n-1}, w_{n-1}, \theta)$ is true if and only if:

$$\sum_{i=0}^{n-1} w_i x_i \geq \theta \tag{3.8}$$

where x_i is equal to 1 if X_i is true, and 0 otherwise. Note that, with unary weights and a threshold equal to n or 1, the WAND operator implements the boolean AND or OR, respectively.

Given a query $q = \{t_0, \dots, t_{n-1}\}$ and a document d, we can apply the WAND operator in the following way. We assume that X_i is true if and only if the term t_i appears in document d, and we take the term upper bound $\sigma_{t_i}(d)$ as weight w_i. The threshold θ has the usual meaning, i.e., the smallest score among the top K documents scored thus far during query processing. Hence the condition WAND $(X_0, \sigma_{t_0}(d), \dots, X_{n-1}, \sigma_{t_{n-1}}(d), \theta)$ evaluates to true if and only if:

$$\sum_{i=0}^{n-1} \sigma_{t_i}(d) \geq \theta \tag{3.9}$$

Assuming that all terms t appear in document d, this inequality corresponds to the pruning condition of Equation (3.4). By using the WAND operator, the authors proposed a DAAT optimisation that can be interpreted as a two-stage evaluation process in which all and only those documents whose WAND evaluation is true (first phase) will undergo a full evaluation where the actual scores are computed (second phase).

Several WAND implementations have been discussed (Broder *et al.*, 2003; Fontoura *et al.*, 2011; Petri *et al.*, 2013). A possible implementation is illustrated in Algorithm 3.2. The algorithm takes as input two arrays of size n: the posting lists `p` to be processed and the corresponding term upper bounds `σ`. Note that they are not required to be sorted since they will be kept sorted though increasing the docids by the algorithm itself (lines 3 and 25). The SortByDocid`(p,σ)` procedure guarantees that the array of posting lists are sorted by increasing docid and that the term upper bound `σ[i]` always corresponds to the posting list `p[i]`.[8] At runtime, the core idea of the algorithm is to evaluate the WAND operator (i.e., the pruning condition) one posting list at a time, accumulating the score of the candidate document in $\sigma_{\mathtt{d}}$ (line 10). As soon as this value exceeds the current threshold (line 11), we have *potentially* identified a *pivot* docid `pivot_id` that could enter the top K documents. The pivot docid undergoes a full evaluation (lines 17–22) and might be included in the current top K results (lines 23–24) only if it is present in all posting lists up to, and including, the list containing the pivot docid. Since the posting lists are sorted by docid, it is sufficient to test the pivot docid with the current posting's docid of the first list `p[0]` (line 16). Otherwise, since the posting lists are sorted by docid, we can safely affirm that the pivot docid is the smallest docid among all posting lists from 0 to `pivot` that could enter the top K results. The docids smaller than `pivot_id` will never be able to accumulate enough term upper bounds to have a chance to beat the current threshold. Unfortunately, we cannot be sure that `pivot_id` will be a candidate, since we do not know yet in which posting lists it appears. Thus, we "backtrack" to the first posting list whose iterator is not on the pivot docid (lines 27–28), and we move its iterator to the `pivot_id` (or further) (line 29) using the `p.next(`d`)` operator. The SwapDown`(p,σ,pivot)` procedure on line 30 restores the docid-sorting of the posting lists by moving the `pivot` posting list and the associated term upper bound "down" to the correct position. Using the `p.next(`d`)` operator means that skipping occurs, and hence the reading and decompression of skipped postings can be avoided. To conclude, note that if no pivot docid can be found (line 13), we are sure that no new document can beat the current threshold, and hence the algorithm can safely terminate.

Figure 3.3 illustrates a few WAND iterations with 4 posting lists. In the example, the current threshold value `θ` is 6.0, resulting from documents in the current top results. The posting list iterators are sorted by current docid. The next pivot id is 22 since neither 2.1 nor $2.1 + 1.3 = 3.4$ are greater than the current threshold, while $2.1 + 1.3 + 4.0 = 7.4$ does exceed the threshold. Since the `p[0]` iterator is not 22, the only knowledge we have so far is

[8] This procedure *does not* sort postings *inside* a posting list, just the array containing the forward-only iterators, as per Section 2.1.

Algorithm 3.2: The WAND algorithm

```
Input   : An array p of $n$ posting lists, one per query term
          An array σ of $n$ max score contributions, one per query term
Output  : A priority queue q of (at most) the top $K$ $\langle$docid, score$\rangle$ pairs,
          in decreasing order of score
WAND(p,σ):
    q $\leftarrow$ a priority queue of (at most) $K$ $\langle$docid, score$\rangle$ pairs,
        sorted in decreasing order of score
    θ $\leftarrow 0$
    SortByDocid(p,σ)
    while true do
        $σ_d \leftarrow 0$
        pivot $\leftarrow 0$
        for pivot $\leftarrow 0$ to $n-1$ do                          // Find list pivot
            if p[pivot].docid() $= \bot$ then
                break
            $σ_d \leftarrow σ_d + σ$[pivot]
            if $σ_d > θ$ then
                break
        if $σ_d \leq θ$ then                                        // No list pivot found
            break
        pivot_id $\leftarrow$ p[pivot].docid()
        if  pivot_id $=$ p[0].docid() then                         // If matching doc pivot
            score $\leftarrow 0$
            for i $\leftarrow 0$ to $n-1$ do
                if p[i].docid() $\neq$ pivot_id then
                    break
                score $\leftarrow$ score $+$ p[i].score()
                p[i].next()
            q.push($\langle$pivot_id, score$\rangle$)
            θ $\leftarrow$ q.min()
            SortByDocid(p,σ)
        else                                              // Else move list up to the pivot
            while p[pivot].docid() $=$ pivot_id do
                pivot $\leftarrow$ pivot $- 1$
            p[pivot].next(pivot_id)
            SwapDown(p,σ,pivot)
    return q
```

that no docid smaller than 22 will have a score greater than the current threshold. Hence, we select p[1] and we advance its iterator to 22, with no reordering of the posting lists since they are already sorted by docid. At the next iteration, our pivot docid is again 22, but we cannot fully score it since we do not know if p[0] will actually contribute to the approximate score of the pivot docid, i.e., we do not know if p[0] contains docid 22. We try to advance the p[0] iterator to 22, but it skips to docid 24, forcing the move of p[0] to the end of the array of iterators. At the next iteration, we are again considering docid 22: its approximate score now is $1.3 + 4.0 + 3.7 = 9.0$, enough for a full processing, and since all posting lists up to the pivot docid include it, docid 22 undergoes a full evaluation, and a potential threshold update. During the full evaluation, the iterators of the involved posting lists are advanced to the next docid (line 22), hence a full reordering of the four lists is mandatory, to correctly select the next pivot docid (that, with the current threshold θ $= 6.0$, will be 24, since σ[2] + σ[0] $= 4.0 + 2.1 = 6.1 >$ θ).

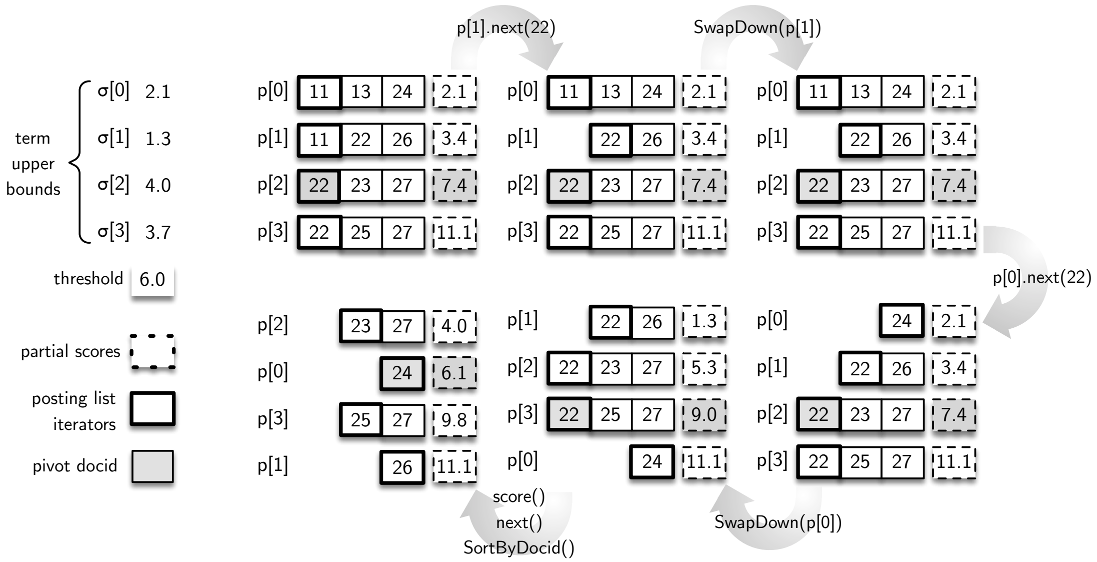


Figure 3.3: How the WAND algorithm processes four posting lists.

Note that the WAND optimisation is safe up to K, since the pruning condition in Equation (3.4) is used to select the candidate docids. Nevertheless, aggressive (i.e., approximate) versions of WAND have been proposed, by substituting θ with $F \cdot \theta$, where $F > 1$ is the *aggressiveness parameter* (Broder *et al.*, 2003; Tonellotto *et al.*, 2013). By using this parameter, we are forcing the new candidate documents to beat the current threshold by a larger quantity.

WAND was designed for disk-based indexes, where disk access is the most expensive cost. For this reason, if a pivot docid is not scored (since there is at least one posting list before the one containing the pivot whose iterator is not yet on or after the pivot docid), the WAND strategy typically chooses to advance just a single posting list, since every iterator advancement is a potential disk access. This is not the case with memory-based indexes, and Fontoura *et al.* (2011) proposed a modified version of WAND, called mWAND, where all the iterators of the posting lists preceding the pivot list are advanced. In this way, the number of pivot selections could be reduced, and consequently, the number of times the posting list array is sorted.

Performances of DAAT, MaxScore and WAND

The most recent performance comparisons of DAAT, MaxScore and WAND appear in (Fontoura *et al.*, 2011) and (Mallia *et al.*, 2017). In the production framework detailed in Section 2.2.3, Fontoura *et al.* (2011) reported the results summarised in Table 3.2. Mallia *et al.* (2017) experimented with a research framework written in C++ with two standard datasets, ClueWeb09, consisting of 50 million English Web pages crawled in 2009, and Gov2, consisting of 25 million `.gov` Web sites crawled in 2004, and two TREC Terbayte track efficiency task topics, namely Trec05 and Trec06, from which they randomly selected $1,000$ queries for different query lengths. Table 3.3 reproduces their main results, for the ClueWeb09 collection and Trec06 experiments only.

Table 3.2: Latency results (in ms) for DAAT, MaxScore and WAND (with speedups and slowdowns) on a small index (up) and a large index (down), for $K = 30$. Adapted from (Fontoura *et al.*, 2011).

	Small index		Large index	
	Short queries	Long queries	Short queries	Long queries
DAAT	0.19	4.55	3.58	26.78
MaxScore	0.17 (1.12×)	2.69 (1.69×)	1.58 (2.27×)	9.32 (2.87×)
WAND	0.21 (0.90×)	5.22 (0.87×)	1.90 (1.88×)	14.08 (1.90×)

Table 3.3: Latency results (in ms) for DAAT and MaxScore (with speedups) for different query lengths, average query times (Avg, in ms) on ClueWeb09, for $K = 10$. Adapted from (Mallia *et al.*, 2017).

	Number of query terms					Avg
	2	3	4	5	6+	
DAAT	60.6	215.9	439.1	686.5	1,270.5	542.5
MaxScore	12.7 (4.77×)	21.3 (10.14×)	27.1 (16.20×)	33.9 (20.25×)	55.0 (23.10×)	32.3 (16.80×)
WAND	14.2 (4.27×)	23.1 (9.34×)	27.3 (16.08×)	37.3 (18.40×)	73.8 (17.22×)	37.2 (14.58×)

Given the differences between the production and research frameworks, the reported latency results and reductions vary as might be expected, in particular since the testbed details in (Fontoura *et al.*, 2011) are not fully disclosed. However, it is clear from both Tables that both MaxScore and WAND provide huge benefits w.r.t. DAAT, and the response time reductions are larger for longer queries (i.e., with 54–61 terms) than for shorter queries (i.e., with 3–5 terms). In particular, from Table 3.2, we can conclude that MaxScore is always better than WAND in terms of speedup, and that WAND is not helpful in reducing the response times of any kind of query on small indexes. From Table 3.3, a detailed analysis of query lengths is carried out and we can conclude that, while WAND and MaxScore exhibit similar speedups for $2-4$ terms queries, WAND quickly becomes less competitive than MaxScore for longer queries. For short queries, mWAND exhibits larger (resp. similar) mean response times than MaxScore (resp. WAND) for short queries, while mWAND is more efficient than both MaxScore and WAND for long queries.

Experiments by Ottaviano *et al.* (2015), on a sample of $5,000$ queries from a realistic query log, showed that MaxScore is always more efficient than WAND, and a similar conclusion is reported by Dimopoulos *et al.* (2013b). Moreover, the pruning capacity of WAND heavily depends on the scoring function used. While most works employ the BM25 scoring function (Robertson *et al.*, 1994), which allows the skipping of more than 90% of all documents for all Ks, Petri *et al.* (2013) have shown that with different scoring functions such as those based on language models 20% of all postings are evaluated for $K=10$ and almost all documents are evaluated when $K=1,000$.

Kane and Tompa (2018) proposed to *prime the threshold* of the max-heap. At indexing time, the K-th highest score contributions in each posting list is pre-computed and stored on disk. Before query processing starts, the max-heap threshold is initialised by taking the greatest K-th score value for all the query terms. They found that using this threshold priming improves the performance of the WAND strategy for large K values.

3.4 Block-based Dynamic Pruning

Thus far, all dynamic pruning optimisations have leveraged a single upper bound for any term involved in query processing for skipping over postings. However, as shown in Figure 3.4, a term upper bound is computed over the scores of all documents in a posting list (left), and it could be larger than the average score contribution (right). This fact limits the potential benefits of skipping over the posting lists, as shown by the shaded areas.

To deal with this, Ding and Suel (2011) proposed to enrich the inverted index data structures with additional information, in a new *block-max index* data structure. Each posting list is sequentially divided in a *block list*, where each block contains a given number of consecutive postings, e.g., 128 postings per block, leveraging the posting blocks used in list-adaptive compression schemes (see Section 2.1.2). For each block, a *block upper bound* is computed, storing the maximum contribution of the postings in the block. This local upper bound can be computed and stored offline, and can then be exploited for improving the efficiency of the existing query processing strategies.

In order to exploit blocks in query processing, the inverted index needs to store additional information (recall Figure 2.1). A possible organisation of such a block-max index is depicted in Figure 3.5.

For each posting list, the lexicon stores the number of its fixed-size blocks,[9] and a pointer to its block list. Every block stores the last docid of the corresponding sequence of postings as well as the block upper bound.[10] Block upper bounds are calculated as the maximum scores of the postings in the block. We will see a block list as a *cursor* over its blocks, analogously to the iterator view of posting lists in Section 2.1.1. We define the following operations on a block list `b`.

- `b.last()` returns the last docid of the current block. If the cursor has reached the end of the block list, `b.last()` returns the special symbol $\perp$. For comparison purposes, the special symbol $\perp$ is considered strictly greater than any other docid.
- `b.score()` returns the block upper bound of the current block.

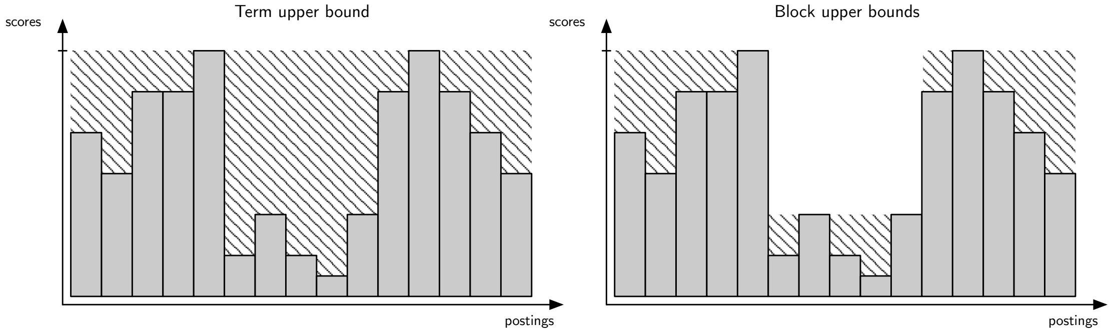


Figure 3.4: Term upper bound (left) vs. block upper bounds (right) on a given posting list. Shaded areas represent the sums of score errors using the upper bounds.

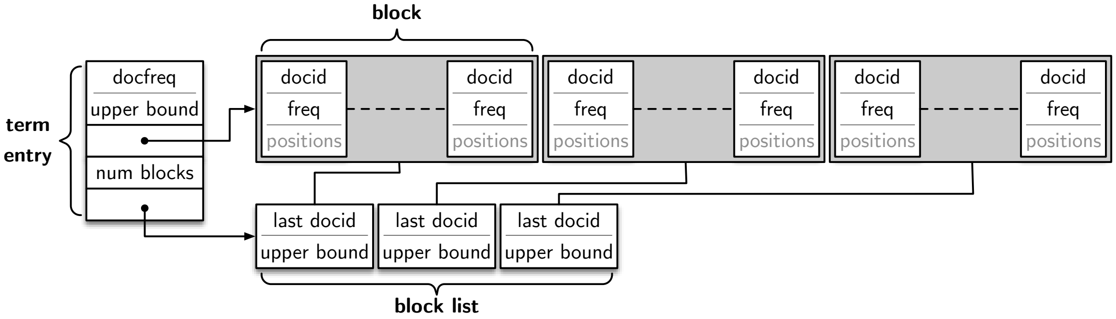


Figure 3.5: A possible layout of a *block-max index*.

[9] We are assuming the size of a block is term-dependent constant.

[10] Other implementations usually store the first docid of the corresponding postings as well (Chakrabarti *et al.*, 2011).

- `b.move(`d`)` moves the cursor to the block containing the document identifier d.

The block upper bounds can be exploited in a number of ways, adapting existing algorithms such as MaxScore and WAND to leverage the new information. The first of such algorithms is Block-Max WAND (BMW), proposed by Ding and Suel (2011), and detailed in Algorithm 3.3.

The algorithm takes as input three arrays of size n, where n is the number of terms to process: the posting lists `p` to be processed with their term upper bounds `σ` and their block lists. The SortByDocid`(p,b,σ)` (lines 3 and 30) and SwapDown`(p,b,σ)` (lines 33 and 40) functions are analogous to the ones in the WAND algorithm (see Algorithm 3.2), guaranteeing that the term upper bound `σ[i]` and the block list `b[i]` always correspond to the posting list `p[i]`. At runtime, the core idea of the algorithm is to avoid the decompression of (blocks of) postings and their full processing by making *shallow* advances through the posting lists by using block lists only, and to make *deep* moves in the posting lists only when the block upper scores sufficiently accumulate to beat the current threshold. At each iteration, the BMW algorithm computes the `pivot` posting list, the `pivot_id` docid and the accumulated (partial) score $\mathtt{σ_d}$ as in WAND, by using the term upper bounds (lines 5–10). The accumulated score $\mathtt{σ_d}$ is also used to check the termination condition. Note that the `pivot` pointer is advanced to include *all* the postings lists whose cursor is on the pivot docid (lines 9–10), while in WAND the pivot pointer included just the posting lists whose upper bounds exceed the current threshold.

Next, the shallow move is performed, by accumulating a new score $\mathtt{σ_b}$ using the block upper bounds (lines 11–17). At the same time, the smallest last docid among the blocks up to the `pivot` is stored in $\mathtt{next_b}$.

If the block score $\mathtt{σ_b}$ beats the current threshold `θ` (line 18), `pivot_id` is a *potential* candidate. If the pivot docid does not match, i.e., if `pivot_id` $\neq$ `p[0].docid()`, this means some posting list iterators up to the pivot are still on docids smaller than the pivot and then we advance one of them to (or right after) `pivot_id` and we reorder the posting lists by increasing docid as in WAND (lines 32–33). Otherwise, we perform a deep move with a potentially *partial* evaluation of the pivot document(lines 20–27) and, as a result, the pivot docid might be included in the current top K results (lines 28–29).

If the block score $\mathtt{σ_b}$ does not beat the current threshold `θ` (line 18), it follows that no document appearing in the blocks can beat the threshold, hence the next pivot docid *could* be strictly greater than $\mathtt{next_b}$ if no other posting list, ignored so far, contributes to the accumulated block score. Hence, $\mathtt{next_b}$ becomes the minimum docid computed between $\mathtt{next_b} + 1$ and the docid of the first posting list whose iterator is not on the pivot docid (lines 35–38), which is guaranteed to be greater than `pivot_id`. Now, we locate a list whose iterator is not on `pivot_id` and we advance it to or right after $\mathtt{next_b}$ and swap down as in WAND (lines 39–40).

Figure 3.6 illustrates a few iterations of the BMW algorithm. The most complex part

Algorithm 3.3: The BMW algorithm

```
Input  : An array p of $n$ posting lists, one per query term
         An array σ of $n$ max score contributions, one per query term
         An array b of $n$ block lists, one per query term
Output : A priority queue q of (at most) the top $K$ ⟨docid, score⟩ pairs,
         in decreasing order of score
BlockMax WAND(p,b,σ):
    q ← a priority queue of (at most) $K$ ⟨docid, score⟩ pairs,
      sorted in decreasing order of score
    θ ← 0
    SortByDocid(p,b,σ)
    while true do
        $σ_d$, pivot ← values updated as in Wand(p,σ) lines 5–12
        if $σ_d$ ≤ θ then                                                  // No list pivot found
            break
        pivot_id ← p[pivot].docid()
        while pivot < $n - 1$ and p[pivot + 1].docid() = pivot_id do
            pivot ← pivot + 1
        $σ_b$ ← 0
        $next_b$ ← $+\infty$
        for i ← 0 to pivot do                                              // Shallow move
            b[i].move(pivot_id)
            $σ_b$ ← $σ_b$ + b[i].score()
            if b[i].last() < $next_b$ then
                $next_b$ ← b[i].last()
        if $σ_b$ ≥ θ then                                                  // If pivot doc would enter
            if pivot_id = p[0].docid() then
                score ← 0
                for i ← 0 to pivot do
                    score ← score + p[i].score()
                    $σ_b$ ← $σ_b$ − b[i].score() + p[i].score()
                    if $σ_b$ ≤ θ then
                        break
                for i ← 0 to pivot do
                    p[i].next()
                q.push(⟨pivot_id, score⟩)
                θ ← q.min()
                SortByDocid(p,b,σ)
            else
                Move list to pivot_id as in Wand(p,σ) lines 27–29                // Deep move
                SwapDown(p,b,σ,pivot)
        else                                                               // Else update blocks
            if pivot < $n - 1$ and $next_b$ > p[pivot + 1].docid() then
                $next_b$ ← p[pivot + 1].docid()
            if $next_b$ ≤ pivot_id then
                $next_b$ ← $next_b$ + 1
            Move list to $next_b$ as in Wand(p,σ) lines 27–29                    // Deep move
            SwapDown(p,b,σ,pivot)
    return q
```

of this algorithm is to keep the block cursors and the posting list iterators aligned when performing shallow and deep moves. In the example, the current threshold value θ is 3.3, resulting from documents in the current top results. The posting list iterators are sorted by current docid. The next pivot id is 22 since $2.1 + 1.3 = 3.4$ is greater than the current threshold. Note that, while WAND would select the second posting list as a pivot, BMW selects the third list (lines 9–10). At the second step (lines 11–17), the block cursor of the first posting list is actually moved, and the block upper bound is computed, resulting in $1.7 + 1.3 + 0.5 = 3.5$. While it is large enough to beat the current threshold, since the first posting list iterator does not point to docid 22, a deep move is performed. At the third step, the pivot docid 22 is evaluated, and the posting list iterators of the first three posting lists are deep moved one step forward at the fourth step, and the posting lists sorted by docid. In the fifth step, assuming that the current threshold has not changed, the pivot docid becomes 24, but its block upper bound is $0.5 + 1.3 + 1.2 = 3.0$, which is not enough to beat the threshold. Since no docid in the current blocks up to the pivot can beat the threshold, the next deep move could be towards the docid $1 + \min\{27, 28, 26\} = 27$. However, since the posting list iterator right after the pivot is on docid 25, the third posting list iterator will be advanced to docid 25, as shown in the sixth step (lines 35–40)

Several versions of Block-Max MaxScore (BMM), the MaxScore variant for block-max indexes, have been proposed in (Chakrabarti *et al.*, 2011; Shan *et al.*, 2012; Dimopoulos *et al.*, 2013b). A possible implementation of BMM is detailed in Algorithm 3.4. The main differences w.r.t. the MaxScore algorithm lie in the early termination controls during the processing of the non-essential lists (lines 18–34). If the non-essential posting lists must be

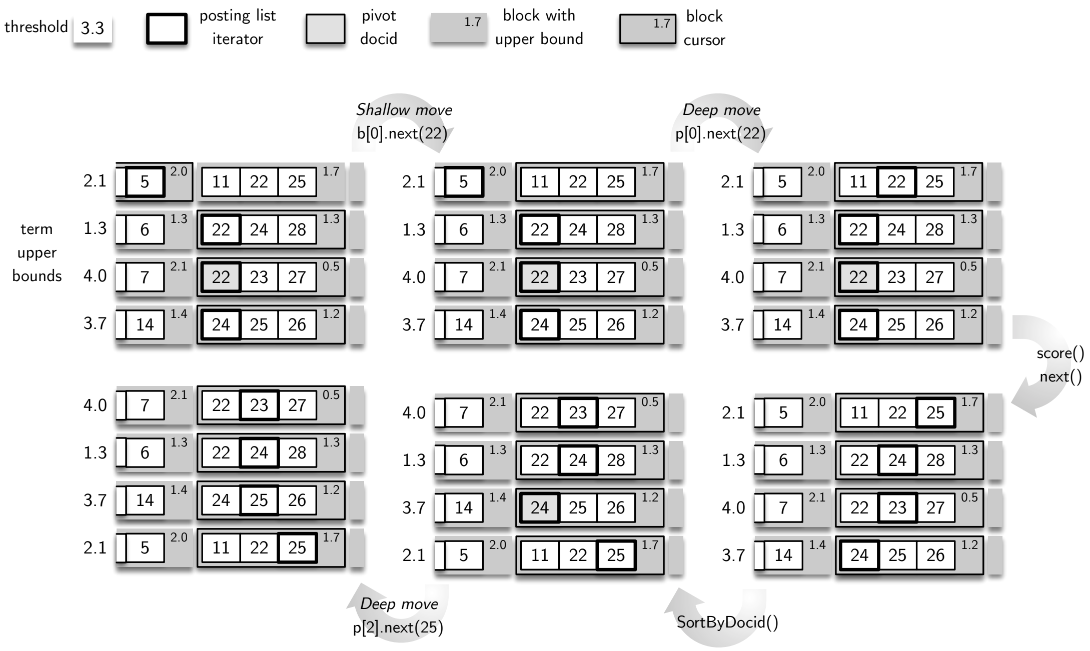


Figure 3.6: How the BMW algorithm processes four posting lists.

processed according to their document upper bound (line 18), a new array bub of (partial) document upper bounds is built, by using shallow moves on the non-essential lists and the resulting block maxscores (lines 19–24). During the processing of a non-essential list, the pruning condition is evaluated by using the block-based document upper bounds bub (line 26).

Algorithm 3.4: The BMM algorithm

```
   Input  : An array p of $N$ posting lists, one per query term
            An array $\sigma$ of $N$ max score contributions, one per query term
            An array b of $N$ block lists, one per query term
   Output : A priority queue q of (at most) the top $K$ $\langle$docid, score$\rangle$ pairs,
            in decreasing order of score
   BlockMax MaxScore(p,b,$\sigma$):
1      q $\leftarrow$ a priority queue of (at most) $K$ $\langle$docid, score$\rangle$ pairs,
          sorted in decreasing order of score
2      ub $\leftarrow$ an array of $N$ max score upper bounds, one per posting list,
           all entries initialised to 0
3      ub[0] $\leftarrow$ $\sigma$[0]
4      for i $\leftarrow$ 1 to $N - 1$ do
5          ub[i] $\leftarrow$ ub[i $-$ 1] $+$ $\sigma$[i]
6      $\theta \leftarrow 0$
7      pivot $\leftarrow$ 0
8      current $\leftarrow$ MinimumDocid(p)
9      while pivot $<$ $N$ and current $\neq \perp$ do
10         score $\leftarrow$ 0
11         next $\leftarrow$ $+\infty$
12         for i $\leftarrow$ pivot to $N - 1$ do                          // Essential lists
13             if p[i].docid() $=$ current then
14                 score $\leftarrow$ score $+$ p[i].score()
15                 p[i].next()
16             if p[i].docid() $<$ next then
17                 next $\leftarrow$ p[i].docid()
18         if score $+$ ub[pivot $-$ 1] $>$ $\theta$ then
19             bub $\leftarrow$ an array of pivot block max score upper bounds, one per non
                      essential posting list, all entries initialised to 0
20             b[0].move(current)                                     // Shallow move
21             bub[0] $\leftarrow$ b[0].maxscore()
22             for i $\leftarrow$ 1 to pivot $-$ 1 do
23                 b[i].move(current)                                 // Shallow move
24                 bub[i] $\leftarrow$ bub[i $-$ 1] $+$ b[i].maxscore()
25             for i $\leftarrow$ pivot $-$ 1 to 0 do                      // Non-essential lists
26                 if score + bub[i] $\leq$ $\theta$ then
27                     break
28                 p[i].next(current)                                 // Deep move
29                 if p[i].docid() $=$ current then
30                     score $\leftarrow$ score $+$ p[i].score()
31             if q.push($\langle$current, score$\rangle$) then                 // List pivot update
32                 $\theta \leftarrow$ q.min()
33                 while pivot $<$ $N$ and ub[pivot] $\leq$ $\theta$ do
34                     pivot $\leftarrow$ pivot $+$ 1
35         current $\leftarrow$ next
36     return q
```

Performance

Ding and Suel (2011) have conducted experiments on a collection of 25 million documents (Gov2), with 2,000 multi-term queries and retrieving the top 10 results per query. Blocks are composed by 64 postings. They reported (see Table 3.4, also validated by Mallia *et al.* (2017)) an average speedup of BMW versus WAND of 2.78×, with extremely high speedups of 5.61× for queries of two terms and 3.70× for queries of three terms. With respect to DAAT, the BMW average speedup is 8.09×. With respect to the evaluated documents, WAND and BMW only evaluate 4.6% and 0.6% of the documents processed by DAAT, respectively. The authors also showed that the performances of WAND and BMW are markedly improved when assigning docids to documents according to the lexicographic ordering of their URLs (Silvestri, 2007) – this is known to exhibit a good clustering and hence a good locality in the posting lists. Thus it follows that documents with high/low scores will cluster in blocks using this docid ordering.

Table 3.4: Latency results (in ms) for DAAT, WAND and BMW (64 postings blocks) (with speedups) for different query lengths, average query times (Avg, in ms) on Gov2, for $K = 10$. Adapted from (Ding and Suel, 2011).

	Number of query terms					Avg
	2	3	4	5	6+	
DAAT	60.0	159.2	261.4	376.0	646.4	225.7
WAND	23.0 (2.61×)	42.5 (3.75×)	89.9 (2.91×)	141.2 (2.66×)	251.6 (2.60×)	77.6 (2.91×)
BMW	4.1 (14.63×)	11.5 (13.84×)	33.6 (7.78×)	54.5 (6.90×)	114.2 (5.66×)	27.9 (8.09×)

Shan *et al.* (2012) reported different results. With a slightly different experimental setting (stopwords removed, ∼3% of single term queries included in averages), the speedup of BMW with respect to DAAT was only 2.38×. They also evaluated an implementation of BMM that, on average, performs better than BMW, with an average speedup of 3.69× compared to DAAT. With respect to the evaluated documents, BMW and BMM evaluate only 0.3% and 3.7% of the documents processed by DAAT, respectively. The higher speedup of BMM even if it scores more documents than BMW is explained by the mechanism governing the next document to process. In the BMM methods, the next docid is selected among the essential posting lists, likely composed by the most important terms, while in the BMW methods, the pivoting is performed among all posting lists for the whole execution of the algorithm.

Dimopoulos *et al.* (2013b) confirmed the BMW experiments in (Ding and Suel, 2011) in terms of average response times, but their implementation of BMM is 1.25 times slower than BMW on average. They reported that BMM outperforms BMW only for queries with more than 5 terms.

Crane *et al.* (2017) reported that the performance gap between WAND and BMW is not as clear as the previous literature suggested. Their tests on a more recent collection of 50 million documents showed that with long queries, i.e., queries with four or more terms, WAND begins to outperform BMW. They explained this behaviour with the additional complex logic required to compute skips in BMW and with the impact of such a logic on the number of cache misses and branch mispredictions.

Variants

While the BMW algorithm is the first proposed algorithm leveraging the block-max index structure, several variants of BMW as well as different uses of blocks have been proposed.

The BMW authors proposed in (Ding and Suel, 2011) a block-max version of BMW for conjunctive processing called Block-Max AND (BMA). They showed performance improvements of BMA over exhaustive AND for queries with two and three terms, while for longer queries the shallow moves make exhaustive AND faster than BMA. They also investigated a layered version of BMW, where each posting list is split in two parts, with high and low impact postings respectively, and each posting list portion is treated as an independent posting list.

In (Shan *et al.*, 2012), the authors proposed a Local BMW variant (LBMW), where the pivot document in BMW is computed by using local block upper bounds instead of global upper bounds. They also proposed a Local BMM variant, quite similar to the BMM algorithm in Algorithm 3.4. Moreover, they also explored BMW and BMM variants taking into account the query-independent document global scores.

Dimopoulos *et al.* (2013b) proposed the BMM-NLB variant of BMM, which skips over *dead blocks*, i.e., blocks that are guaranteed to have a block upper bound over all posting lists smaller than the current threshold, similarly to the approach in (Chakrabarti *et al.*, 2011). In the same paper, a hierarchical organisation of blocks is proposed, with a complex query processing algorithm inspired by branch-and-bound techniques, called HIER. The authors also investigated a different organisation of blocks. While the previous works considered blocks as a contiguous sequence of postings (*posting-oriented blocks*) leveraging the posting grouping performed in block-based compression, in this paper the whole docid space, i.e., the set of all docids, one per document in the collection, is partitioned into blocks (*docid-oriented block*). For example, with block size 1024, all documents with docid from 0 to 1023 will end up in one block, docids from 1024 to 2047 in another block, and so on. This will result in blocks with different sizes in the same posting lists. They also proposed to manually tune the docid block size depending on the posting list length, to limit the space occupancy of the additional information required.

Dimopoulos *et al.* (2013a) illustrated an optimised implementation of docid-oriented blockmax indexes exploiting the SIMD processing capabilities of modern CPUs and ad-hoc caching policies. Rojas *et al.* (2013b) and Rojas *et al.* (2013a) presented parallel and distributed implementations of BMW, respectively, while Daoud *et al.* (2016) discussed how BMW can be implemented on top of a tiered inverted index, extending their previous results from (Rossi *et al.*, 2013).

Mallia *et al.* (2017) introduced a refinement for BMW that uses variable-size blocks, rather than constant-sized blocks, called Variable BMW (VBMW). They considered the problem of deciding the block partitioning of a posting list as an optimisation problem, which maximises how accurately the block upper bounds represent the underlying scores, and described an efficient dynamic programming approximate solution. Their experiments showed that VBMW outperforms BMW with a speedup of roughly 2×. Moreover, they proposed a compression scheme for block upper bounds based on quantisation. The compressed VBMW strategy obtained a space reduction of 50% with only a minimal speed degradation w.r.t. the uncompressed VBMW counterpart.

Conditional Skipping

Bortnikov *et al.* (2017) proposed to add a new operation to the postings' APIs discussed in Section 2.1.1, namely the *conditional skip*:

- `plist.next(`d`, `τ`)` advances the iterator forward to the next posting with a document identifier greater than or equal to d, while ensuring that all skipped postings have a score lower than τ. If the current posting's docid is greater or equal to d, the iterator is left unchanged. If d is greater than the docid of the last posting in the list, `plist.next(`d`)` returns the end-of-list marker $\perp$. More precisely, this operator advances the posting list iterator `plist` to the first posting with a docid greater than or equal to d such that either `plist.docid()` $\geq d$ or `plist.score()` $\geq \tau$.

The conditional skip operator can be used to improve the performance of DAAT processing and its optmisations, MaxScore, WAND and BMW. Two simple examples of the conditional skip behaviour are reported in Figure 3.7. In both examples, there are two posting lists `p[0]` and `p[1]`, whose iterators point to the postings of docids 11 and 40, and the current threshold is equal to 4.0. Any docid in the range $[11, 40)$, appearing in the posting list `p[0]`, can be safely skipped (top example in Figure 3.7) unless one of the docids in this range has a score greater than the current threshold (bottom example).

Bortnikov *et al.* (2017) proposed different implementations of the conditional skip, and modified the DAAT, MaxScore, WAND and BMW strategies to use it. Their experiments on the ClueWeb09 B dataset showed a reduction in the document scoring overheads, and also in query processing times.

3.5 Limitations of Dynamic Pruning

Several of the dynamic pruning techniques discussed above are based on additional statistics calculated upon the inverted index, such as the global or block-based upper bounds $\sigma_t(q)$ for each term t. Obtaining these involves the scoring of every posting in each term's posting list. This is not an expensive process when conducted offline, before an index is deployed (and noting that its an embarrassingly parallelisable process). However, the exact pre-calculation of term upper bounds $\sigma_t(q)$ has some disadvantages, in that it is sensitive to changes in the ranking function scores (e.g., the used weighting model). This may happen in a number of cases:

1. Adaptation of the weighting model, or its hyper-parameters (e.g., the parameter tuning of the search engine);
2. Adaptation of the index, e.g., adding or removing documents, thereby changing the global statistics of the index (number of documents, average document length, document frequency);
3. Adaptation of a given term's posting list, e.g., changing documents, thereby changing the statistics of the term (e.g., term frequency).

We note that Macdonald *et al.* (2011) and Macdonald and Tonellotto (2017) proposed approximate (less-tight) upper bounds. These can mitigate some of the disadvantages noted above, but could also result in reduced efficiency.

Assuming that exact upper bounds are necessary, given the efforts in scoring all postings

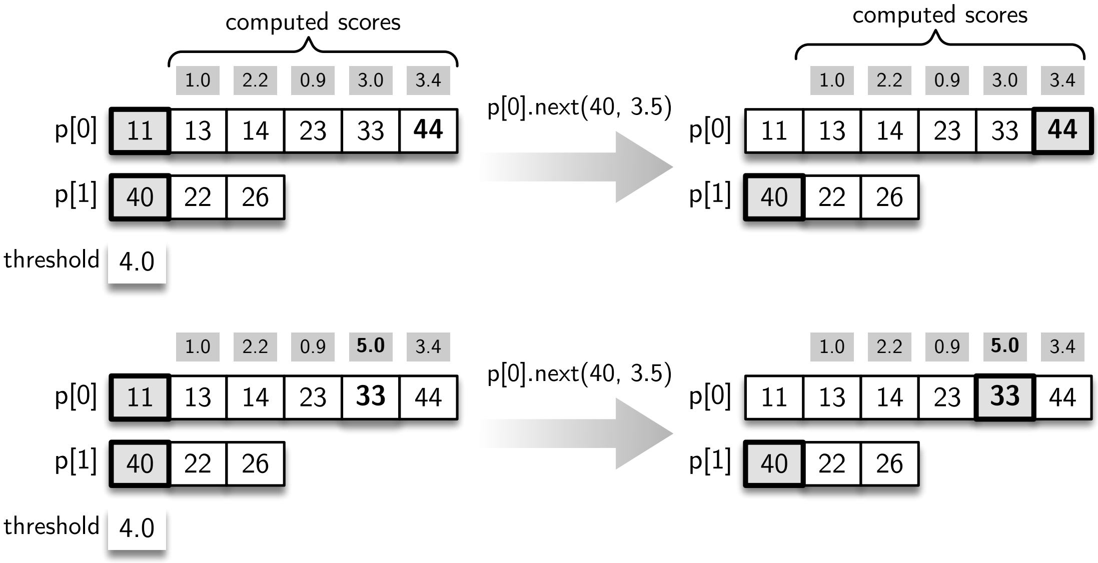


Figure 3.7: A simple example of the *conditional skip operator*.

in the inverted index, a notable question would be if that more efficiency benefits can be sought from this extra processing, for instance by rewriting the inverted index into a more efficient form, that focuses upon documents that are more likely to be retrieved. This will be the focus on Chapter 5.

3.6 Summary

This chapter introduced dynamic pruning techniques for attaining efficient retrieval. After an introduction to early termination and various core concepts, we presented, for completeness, a full range of TAAT optimisations, very common in the early IR systems but with almost no known practical deployment nowadays. Then we presented with clear examples and pseudo-codes the DAAT optimisations, currently used in most IR systems: MaxScore, WAND and BMW. Their definitions have significantly improved the efficiency of many deployed Web search engines. In the next chapter, we will discuss recent trends in the development and applications of query efficiency predictors for dynamic pruning (Section 4). Indeed, some of these applications have demonstrated remarkable benefits in increased efficiency or reduced server resource utilisation, with corresponding energy reductions.

Most of the techniques discussed within this chapter are concerned with inverted index layouts where posting lists are docid ordered, as used by at least one commercial search engine (Dean, 2009). In Chapter 5, we will address other techniques that alter the inverted index, such as index layouts with alternative orderings that ensure that documents more likely to be retrieved are identified earlier during retrieval – such techniques generate a different set of optimisations, which we describe in detail.

4 Query Efficiency Prediction for Dynamic Pruning

In this chapter, we discuss a new technique gaining attraction for a number of applications, namely *query efficiency prediction* (QEP) (Macdonald *et al.*, 2012d). In particular, as will be clear from the experimental results reported in Chapter 3 above, the execution time of different queries can vary. Moreover, different techniques can exhibit different speedups for different queries. Hence, it is intuitive that different query processing techniques might be applied for different queries, or tuned differently (for example varying the aggressiveness, F, of WAND or BMW on a per-query basis). Obtaining accurate estimations of the response time of the search engine for a query allows the deployment of such per-query optimisations.

Aside: We emphasise the difference between query *efficiency* prediction and query *performance* prediction (QPP). In QPP, the aim is to predict the likely effectiveness of the results set for the query (e.g., in terms of MAPor NDCG) – this can be estimated by examining *pre-retrieval* statistical properties of the query terms (He and Ounis, 2006), i.e., before retrieval commences, or *post-retrieval*, by examining the retrieved documents (Cronen-Townsend *et al.*, 2002). In 2010, Carmel and Yom-Tov reviewed many of the existing query performance prediction techniques available at that time. In contrast, the task of query efficiency prediction, described in this chapter, is inherently pre-retrieval, in that once retrieval has been performed, we will know the actual, exact response time.

Briefly, the main idea behind query efficiency prediction is to estimate how long an unseen query will take to be processed, before it has executed. The main factors that can affect how long a search engine will take to respond to a given query are, for instance, the number of query terms, or the length of the posting lists of the query's constituent query terms. For example, this is true for both DAAT and TAAT traversals. However, dynamic pruning adds a further dimension to this, as different queries may suffer different amounts of pruning, and the number of documents being retrieved affect the response times of the search engine. This chapter specifically discusses how the QEP approaches could be deployed to assist the optimisation of dynamic pruning approaches.

At this juncture, it is important to point out that impact-sorted index layouts, as might be processed by a SAAT technique (further detailed in Chapter 5) do not require query efficiency prediction, as the time to execute a query can be much more accurately predicted.

On the other hand, to predict the query execution times of the DAAT query evaluation techniques, Macdonald *et al.* (2013b) resorted to a machine-learned framework, based on term-level statistics (including, among others, the length of posting lists, and the number of high-scoring documents within those posting lists), as well as *aggregators*, such as max, min, mean, to compose scalar values across the various terms in a query. This chapter will provide an overview of query efficiency prediction, which can be made *prior* to the execution of the query (Section 4.1), or *while* the query is executing (Section 4.2). Moreover, query efficiency prediction permits the development of approaches that can respond to the duration that a query is expected to take, which we describe further in Section 4.3. In particular, we will provide various insights into five applications of query efficiency prediction. In the following, we first describe various implementations of query efficiency prediction used in the literature.

4.1 Implementations of Query Efficiency Prediction

The task of predicting the execution time of a query before it executes is itself not easy to accurately achieve. As mentioned above, there are a number of factors that can affect the execution time of a query, for instance the length of the query in terms, or the length of the constituent posting lists of those terms. These factors are adequate to predict accurately the execution time of the TAAT and DAAT strategies.

However, while the above is certainly true for the response times of an IR system with a full exhaustive scoring of the postings lists, for dynamic pruning approaches such as WAND, the pruning behaviour means that not all postings in a posting list will be scored, nor even decompressed. Moreover, not all queries benefit equally from dynamic pruning. For example, consider two queries with two query terms each: In the first query, one term is considerably less frequent in the collection than the other, and hence is dominant in the document scoring (i.e., it has a large IDF component); For the second query, the two query terms have approximately equal IDF. For the first query, dynamic pruning will usually avoid the scoring of many postings for the second query term. On the other hand, for the second query, it is likely that the occurrences of both query terms will need to be scored. Macdonald *et al.* (2012d) described this observation as variations in the *pruning difficulty*. The aim of query efficiency prediction is to empirically estimate the pruning difficulty.

The main approach taken by the literature (Macdonald *et al.*, 2012d; Jeon *et al.*, 2014) is to approach QEP as a supervised regression problem. Queries are represented by a number of features, and a regression model is obtained by learning from a past history of queries and their execution times. Given the likely applications of QEP, it is important that the calculation of such features is extremely timely, and therefore based on statistics that may be recorded for each term in the lexicon data structure. This then naturally leads to the framework proposed by Macdonald *et al.* (2012d), in that term-level statistics, such as posting list length, are aggregated for each term in the query, using aggregation functions such as max, sum, etc, into features for learning the model. More formally, let $f_{ij}(q)$ be a feature defined for query q using term statistic $s_j(t)$ and aggregation function A_i (max, sum, variance):

$$f_{ij}(q) = A_i(\{s_j(t), \forall t \in q\}). \tag{4.1}$$

Different term-level statistics have been examined. Clearly, the number of postings for a term t, denoted N_t, should be used, as this is the simplest predictor, useful for exhaustive approaches, and also for queries where no dynamic pruning is possible (i.e., $\sum_{t \in q} N_t$). Moreover, for WAND, the minimum of N_t across the query terms is indicative of the minimum number of postings that will definitely be scored, in the presence of excellent pruning conditions. Other features examined the score distributions within each term's posting list (e.g., term upper bound, arithmetic mean of the scores of document in a posting list or number of postings within 5% of the maximum score). These help to indicate the likely pruning difficulty. Learning a supervised approach using these features, combined with a linear regression learner, have demonstrated over 0.9 Pearson's correlations for queries of lengths $2 - 5$ (Macdonald *et al.*, 2012d).[1]

[1] As no dynamic pruning is possible by WAND for single-term queries, the prediction of the execution time of single-term queries is trivial.

Jeon *et al.* (2014) enhanced the earlier work on query efficiency prediction in two ways. Firstly, by recognising that for commercial search engines, the queries that are expected to exceed the service level agreement (SLA) are those that really matter. Indeed, their work targets the Bing search engine by aiming to address an SLA of 100 ms for 99-th percentile response time (i.e., only one query in 100 can exceed 100 ms). QEP can hence be formulated as a classification task: predict accurately those queries that will exceed such a *tail* latency target. Secondly, they proposed small improvements to enhance QEP, such as recognising that the minimum aggregation function is useful for QEP.[2] Moreover, when dealing with queries within the context of the Bing search engine, advanced query features such as relaxations (e.g., "facebok OR facebook") are considered. They reported that the new features lead to large benefits in tail query classification precision and recall. Moreover, they experimentally show that gradient-boosted regression trees provide better results than linear regression.

Returning to our example two-term queries with varying pruning difficulties, even queries where both query terms t_1 and t_2 have roughly equal IDF scores (i.e., $N_{t_1} \simeq N_{t_2}$) can result in a varying number of postings being scored, depending on the *correlation* between the terms: if the two terms occur independently, then the overlap between the docids in their posting lists will follow a random distribution. On the other hand, if the two terms are correlated (e.g., they frequently co-occur in the collection), then the docids in their posting lists will be similar. An analytical method of query efficiency prediction was proposed by Wu and Fang (2014), who noted that if two query terms t_1 and t_2 are independent, then the expected number of documents containing both terms is $\frac{N_{t_1}}{N} \times \frac{N_{t_2}}{N} \times N$. This can be generalised, such that when the two terms are not independent and $N_{t_1} \leq N_{t_2}$, the number of documents containing both terms, $A(t_1, t_2)$ is approximated as follows:

$$A(t_1, t_2) = \frac{N_{t_1}}{N} \times \left(\frac{N_{t_2}}{N}\right)^{\delta} \times N, \tag{4.2}$$

where δ is a parameter used to control how related are the terms t_1 and t_2. In general, it is impossible to store the frequency of co-occurrence of arbitrary terms in a corpus without resorting to accessing an inverted index, so the authors “arbitrarily” assume that $\delta = 0.5$ is a good value in the general case. By using this approximation, Wu and Fang (2014) showed how to estimate new dynamic pruning features for a given query, such as the number of blocks to be accessed and decompressed, the number of documents to be processed and the number of postings to evaluate when processing the query. Overall, this proposed analytical approach for QEP showed promise in decreasing errors in the predicted execution times compared to the machine learned approach of Macdonald *et al.* (2012d).

[2] This aggregator was missing in (Macdonald *et al.*, 2012d).

4.2 Delayed Query Efficiency Prediction

Depending on the application of QEP, it may be possible to delay the calculation of the QEP until the query has been running for a small amount of time. At this stage, the matching algorithm might have access to certain statistics that allow an accurate depiction of the likely execution time of the query. Such an approach was proposed by Kim *et al.* (2015) to reduce the *extreme* tail latencies, i.e., the 99.99-th percentile, of query servers in a commercial search engine by parallelisation (see Section 4.3.3).

The main idea is to allow queries to be executed for an empirically tuned short amount of time (e.g., 10 ms). This preliminary processing has two main benefits: firstly, short queries that can be processed in less than 10 ms do not need either efficiency prediction nor parallelisation. Secondly, such preliminary execution can provide *dynamic features* affecting query execution that can be used for query efficiency predictions. Such features include statistics on the dynamic score distribution observed at runtime, the number of processed documents and the average time to match two consecutive documents. The features may also include estimates on the co-occurrence frequency of query terms. Indeed, the independence of the query terms cannot be well approximated in prior to execution of a query (see $A(t_1, t_2)$ above). This is because it is not possible to record the co-occurrence frequency of arbitrary terms in a space-efficient manner without resorting to the inspection of an inverted index. However, if term occurrences are assumed to be distributed uniformly through the index (they may not be – often some orderings of docids cause a "clustering" effect of term occurrences – see Section 2.1.2, page 19), then dynamic estimates obtained as the query executes may be sufficiently good enough to estimate accurately the expected execution time of the query.

The experiments in (Kim *et al.*, 2015) showed that the new dynamic features provide important information on predicting the query execution times of long queries, coupled with aggregators to summarise term-level information at query-level. In particular, learned models classifying extreme tail latencies obtained a great boost in prediction precision over static features without delayed prediction.

4.3 Query Efficiency Prediction Applications

We now describe the applications of QEP, namely on-the-fly adaptations of the configuration of the search engine on a per-query basis. The underlying aim of these applications is to enhance the query processing component of the search engine, and to reduce the overall query response times. Many of these applications take into account the Service Level Agreement (SLA) that the search engine targets, e.g., a constraint on the duration of mean or long-running[3] query response times. In the following, we provide insights into five different applications of QEP– in each case, such applications are designed to increase the efficiency of the search engine over different elements of the query processing component, which will have resulting benefits upon server capacity and energy savings, namely:

- **Selective Pruning** (Section 4.3.1): Selective pruning is concerned with adjusting the F tradeoff parameter of a dynamic pruning technique for queries that are predicted to take a long time, or adjusting K, the number of documents retrieved by the dynamic pruning stage, which are then re-ranked by the application of a learned model in a subsequent stage (covered further in Chapter 6).
- **Selecting** the **replicated** query server likely to be the least busy (Section 4.3.2): Replicated query servers in a distributed search engine may not be equally busy, due to the varying volume of queries being received. Predicting the execution time of queries allows us to employ effective scheduling strategies leveraging such information, such as the shortest job first strategy, instead of the simple first-come first-served approach.
- **Selective Parallelisation** (Section 4.3.3): As queries take varying amounts of time to execute, long-running queries can be spread (or parallelised) among multiple CPU cores, to ensure that the SLA constraints are met.
- **Selective query re-writing** (Section 4.3.4): Queries may be rewritten internally before execution, for instance through the application of proximity operators or stemming, if the rewritten query is more likely to complete within the constraints of the SLA.
- **Selective CPU frequency scaling** (Section 4.3.5): Since users can hardly notice response times that are faster than their expectations, a CPU can process queries at the lowest frequency respecting the user-defined deadlines, in such a way to save energy and reduce the operational costs of Web search engines.

In the following, we describe the aforementioned five applications in detail, and survey a number of corresponding approaches from the literature.

4.3.1 Selective Pruning

As discussed in chapter 6, a dynamic pruning strategy is often considered as input to a (series of) refined ranking stage(s), which will re-rank those K results to increase effectiveness. This is achieved by calculating additional features, and applying a learned model obtained from a learning-to-rank technique.

The effectiveness of the K documents ranking has therefore a role in the effectiveness of

[3] For instance, some works describe SLAs in terms of tail response times, such as 95-th percentile response time.

the final ranking presented to the user – for instance, approximate or set-safe retrieval[4] may not have a considerable impact on user satisfaction, as the re-ranker will still be able to identify highly relevant documents within the initial K documents.

With this in mind, Tonellotto *et al.* (2013) investigated *selective pruning*, where the number of documents K and the pruning aggressiveness F are varied on a per-query basis (recall, from page 51 that $F > 1$ makes WAND approximate in nature, degrading effectiveness in favour of increased efficiency). In doing so, they examined two intuitions, aiming to enhance efficiency while maintaining effectiveness:

- Queries that are predicted to be easy should be targeted for more aggressive pruning. For such queries, the relevant documents will be highly scored, and hence, sufficient recall is obtained even when aggressive pruning ($F > 1$) or smaller K values are used.
- Queries that are predicted to take a long time to execute should be targeted for more aggressive pruning. By targeting such inefficient queries (which typically have long posting lists), we can directly benefit efficiency, while aiming for not markedly damaging effectiveness. Indeed, applying more aggressive pruning will skip more of the scoring of the less informative query terms with long postings lists, which are less likely to change the retrieved documents.

Hence, their experiments compared the application of a per-query pruning decision mechanism for varying K and F that compared the use of QPPs vs. QEPs. Indeed, their experiments using the ClueWeb09 B test collection showed that using QEPs resulted in marked efficiency improvements (decreasing mean response time by 36% and the response time experienced by the slowest 10% of queries by 50%) while ensuring that effectiveness was maintained.

Later, Broccolo *et al.* (2013) went further by improving selective pruning, by observing that the appropriate aggressiveness for a query should be determined not just by considering the current query, but by also considering the current load of the machine, i.e., other queries currently *queued* or being processed by that machine.

To explain this further, consider that a given query server within a search engine can only process a fixed number of queries concurrently, e.g., based on the number of CPU cores, and that any other queries are queued, in a first-in first-out basis, until they can be processed. Furthermore, consider that the search engine has a SLA for queries being executed, say within 100 ms. Queries that spend time in the queue, waiting to be executed, are increasingly moving toward their 100 ms deadline without having made any progress. Hence, the available time to process such queries is decreasing. To address this, they took the natural step of using the predicted execution time of the query *and* the queries following it in the queue into account when determining the aggressiveness for a given query.

The work of Broccolo *et al.* (2013) considered four strategies that might be used when

[4] See page 37 for definitions.

determining the pruning aggressiveness for a query:

- *Manic*: All queries are processed as quickly (aggressively) as possible, leading to small response times, but also to degradations in the effectiveness of all queries.
- *Perfectionist*: All queries are processed as slowly as possible, using a safe-to-rank strategy, leading to larger response times, but without degradations in their effectiveness.
- *Selfish*: Each query is processed allowing as much time as possible to permit its processing deadline to be met, making use of QEPs.
- *Altruistic*: This approach aims to be fairer, by aggressively pruning queries, thereby aiming to ensure queries in the queue can meet their deadlines. This method works by using QEP to compute the time for the entire queue of queries to be completed. Any slack time is spread fairly across all queries.

Their experimental setup used the ClueWeb09 B corpus, partitioned across ten index shards. Queries were answered using the TAAT Continue method. Their results showed that at a workload of 40 queries per second, the Altruistic approach is able to meet a deadline of 0.5 seconds for 90% of queries whilst still attaining high effectiveness. Overall, selective pruning is a promising application of QEPs, where the configuration of the dynamic pruning technique is performed on a per-query basis.

4.3.2 Selecting among Replicas in a Distributed Retrieval Engine

As highlighted by the work of Broccolo *et al.* (2013) described above, in some search scenarios, queries will arrive faster than they are processed. This mandates a need for queuing queries as they arrive.

Figure 4.1 provides a graphical illustration of the conceptual architecture of a distributed search engine, where a given index shard is replicated R times. Queries arrive at a central *broker*, which *schedules* these queries across the replicated query servers.

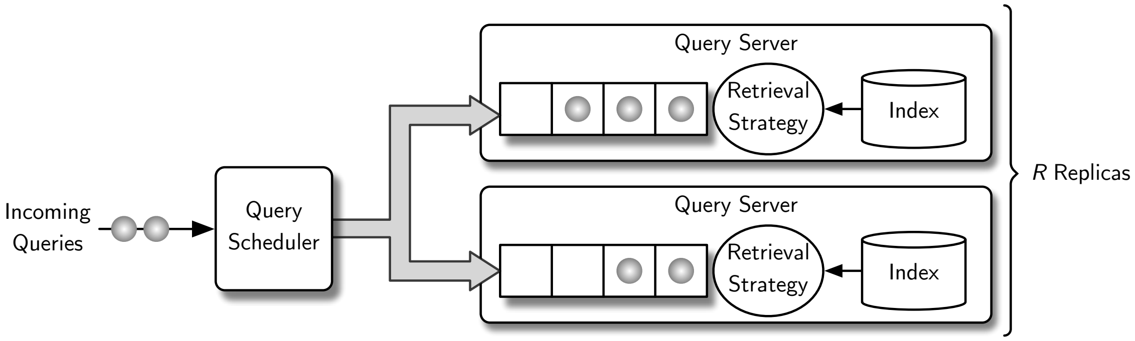


Figure 4.1: Conceptual architecture for a replicated index shard with R replicated query servers, where each replica has its own queue of queries to process.

From Figure 4.1, it is easy to see that the next query would likely be directed to the second replicated query server, as this server has the smallest queue of waiting queries. Such a scheduling choice would be optimal if all queries took the same time to execute. However, as queries vary in their execution times, such a scheduling decision would lead to queries taking longer to execute, and potentially exceeding their SLA.

In (Macdonald *et al.*, 2012d), the authors demonstrated an application of QEPs for scheduling queries within such a replicated architecture. In particular, queries are received by a query scheduler, which computes the predicted execution time of the query. The scheduler can then schedule the next query to the node with the shortest predicted execution time for the queries currently queued. This prevents queries being starved behind long-running queries.

Their simulated experiments compared queue length-based scheduling (where a query is routed to the server with the shortest queue) with predicted execution time-based scheduling, based on QEPs, and a best case scheduling where the actual execution times were known in advance (i.e., using perfect QEPs). The results demonstrated the usefulness of the predicted execution time-based scheduling, with a 22% reduction in the time queries spent waiting to execute, compared to queue length-based scheduling. Overall, these experiment demonstrated a further potential application of QEPs to reduce query response times, in particular within a distributed search engine setting.

4.3.3 Selective Parallelisation

The execution time of queries by some retrieval strategies can benefit from the use of multiple CPU cores. Jeon *et al.* (2013) described an approach deployed in the Bing search engine where "chunks" of posting lists are assigned to a work queue. Under normal, single-threaded execution, chunks are processed sequentially resulting in a normal-esque retrieval process, but where the final scores of the documents in the global heap are updated at the end of each chunk. For some queries, chunks can be assigned to different execution threads. The use of thread-local heaps reduce inter-thread contention in accessing the global heap.[5]

Through experiments using a stream of Bing queries, their approach was shown to attain a speedup of 4×, using 6 CPU cores, for the 5% of queries that take the longest to execute. However, not all queries should be parallelised, as this would be an inefficient use of resources, which could be used to process other queries. Jeon *et al.* (2013) examined two heuristics for making parallelisation decisions: system load, i.e., number of waiting queries; and a parallelisation efficiency profile, which estimates if a query would likely benefit from parallelisation.

Later, Jeon *et al.* (2014) extended their deployment of parallelisation to use QEP,

[5] Later work by Jeon *et al.* (2014) clarified that query processing takes place in a DAAT fashion, using early-termination.

particularly targeting queries exhibiting 'tail latencies', i.e., high-percentile response times that might not meet the Bing service level agreement. Such queries were *selectively* identified using efficiency predictions. Comparative experiments to the earlier adaptive approach (from (Jeon *et al.*, 2013)) demonstrated that by using QEP for selective parallisation, a 99-th percentile response time of 100 ms was achieved for query loads of upto 750 QPS. In contrast, using the adaptive approach, the 99-th percentile response time exceeded 100 ms at the fairly low-load rate of 300 QPS. This was due to the adaptive approach incorrectly parallelising many queries that would have completed anyway within 100 ms. The experiments reported in Jeon *et al.* (2014) demonstrated a noteworthy benefit to the Bing search engine, with the predictive approach increasing the server throughput by 50%. Such a throughput increase is equivalent to a potential saving of one-third of production servers, constituting a marked energy cost reduction in a large-scale system with tens or hundreds of thousands of servers.

More recent improvements reported by Kim *et al.* (2015) showed the usage of delayed prediction (discussed separately above in Section 4.2), and a deployment on heterogeneous CPU architectures. In particular, Kim *et al.* (2015) found that they could schedule queries with longer predicted execution times onto faster, more expensive CPU cores.

Overall, the findings reported in the literature for this application demonstrate the potential for QEP to have substantial efficiency benefits to a commercial search engine, with resulting energy savings.

4.3.4 Selective Rewriting

Many approaches have been proposed to rewrite the user's initial query, to encapsulate stemming (Peng *et al.*, 2007), common query reformulations (Jones *et al.*, 2006), or the inclusion of complex "proximity" operators that boost the scores of documents where the query terms occur closely together (Bendersky *et al.*, 2010; Metzler and Croft, 2005). However, many such techniques degrade the efficiency of the search engine, resulting in a longer, more complex query that takes longer to execute.

To address this issue, Macdonald *et al.* (2017) described the application of QEPs to facilitate effective and efficient rewriting of the user's query. Indeed, they noted that the application of learned models means that techniques such as proximity can be applied either within the first-pass retrieval (i.e., during the WAND or BMW retrieval processes), or later as a feature calculated on the top K documents. Therefore, there is a choice: invest execution time in the first retrieval phase, by using a complex query formulation, which may allow smaller K sets to be used; or use a simpler query with a larger K. Figure 4.2 illustrates the available space of solutions for each query, where the size of the circles illustrates the likely execution time. In each case, choosing strategies that lie far from the origin results in decreased efficiency, and, hopefully, increased effectiveness.

Macdonald *et al.* (2017) built upon the earlier work of (Tonellotto *et al.*, 2010) that deploys QEPs. Here, these QEPs are repurposed to eliminate strategies that cannot execute within the constraints of the SLA. Of the remaining strategies, the one that demonstrated the highest mean effectiveness on previous training queries is then applied to execute that query. From their experiments using the ClueWeb09 B dataset, they reported a 49% decrease in mean response time, and 62% decrease in tail (95-th percentile) response time, without significant degradations in effectiveness.

As an aside, we note that a further contribution of Macdonald *et al.* (2017) is the derivation of QEP estimates when the query contains complex operators (e.g., `#uw`λ, which counts the frequency of query terms within a window of n tokens in each document). Such operators are commonly used to implement proximity weighting models – complex operators are further described in Section 2.2.4.

In summary, query rewriting is an important aspect of modern search engines, which must account for the efficiency costs as well as the effectiveness benefits. In this application, by deploying QEP, the authors showed that they could select among possible rewritings of the query those that benefit both effectiveness and efficiency.

4.3.5 Selective CPU Frequency Scaling

Web search engines perform distributed query processing on computer clusters composed by thousands of computers and hosted in large data centers (Cambazoglu and Baeza-Yates, 2015). While such facilities enable large-scale online services, they also raise economical and environmental concerns. Indeed, a large-scale data center – like those used by Web

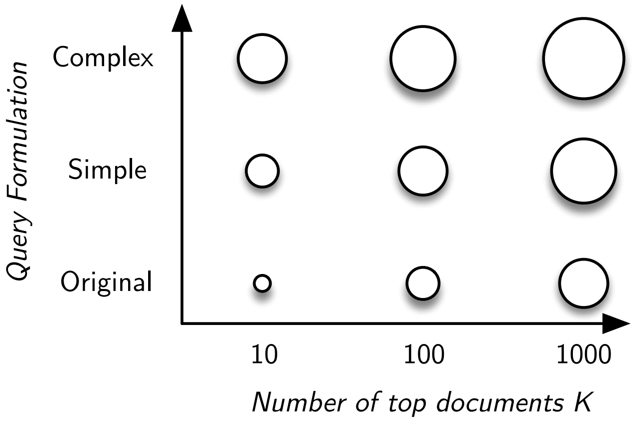


Figure 4.2: Illustration of the search space of strategies to handle a query, varying both K and the complexity of the rewritten query.

search engines – can draw tens of megawatts of electricity to operate and it can cost several millions of US dollars per year in terms of energy expenditure (Greenberg *et al.*, 2008). CPUs are the most energy consuming component in servers dedicated to query processing, accounting for 40% of the total energy consumption when a server is idle and for 66% of the total energy consumption when it is fully utilised (Barroso *et al.*, 2013). Catena *et al.* (2015) proposed to use Dynamic Voltage and Frequency Scaling (DVFS) technologies to reduce the CPU energy consumption of a query server. Leveraging the fact that users can hardly notice response times that are faster than their expectations (Arapakis *et al.*, 2014), Catena and Tonellotto (2017) postulated that Web search engines should not process queries faster than a user's expectations and, consequently, they proposed the Predictive Energy Saving Online Scheduling (PESOS) algorithm. This algorithm considers the latency requirement of queries as an explicit parameter, and tries to process queries no faster than required. In doing so, the CPU energy consumption is reduced, while respecting the query latency constraints. PESOS bases its decision on frequency-dependent query efficiency predictors. Firstly, a modified QEP predicts the number of scored postings for a query with a given number of terms, independent of the underlying CPU features, then a single-variable linear predictor forecasts the processing time of a query composed by a given number of terms at a certain CPU frequency through the estimated number of its scored postings. PESOS leverages such predictions to select the most appropriate CPU frequency to process a query by its deadline, e.g., 500 ms, on a per-core basis.

The experimental evaluation of PESOS upon the ClueWeb09 B collection and the MSN 2006 query log showed that PESOS can reduce the CPU energy consumption of a query processing server from 24% up to 48% when compared to a high performance system running at maximum CPU core frequency, depending on the required latency. Overall, this QEP application demonstrates that by considering the execution characteristics of a query, significant energy savings can be made on the servers executing the query.

4.4 Summary

Overall, query efficiency prediction has been shown to be a promising concept for the on-the-fly adaptation of the search engine's configuration. However, thus far, QEP is only applicable to DAAT query processing - i.e., only in DAAT do the query response times for queries exhibit such variance. Moreover, such DAAT techniques may take a long time to answer a query exhaustively (and safely), and not have increased the effectiveness compared to a more aggressive, unsafe configuration. In contrast, in the next chapter, we describe impact-sorted indexes, which due to their nature of focusing on higher value documents earlier in the posting lists, lead to a less degraded effectiveness if the scoring process is terminated earlier. While this negates the need for QEP, impact-sorted indexes also have other disadvantages, as we discuss in the next chapter.

5 Impact-Sorted Indexes

Despite the best attempts of dynamic pruning techniques to avoid scoring non-relevant documents, or those that will never be retrieved, for a particular query, there are documents in the index that will never be retrieved for *any* query. Hence, a natural question arises as to whether we can eliminate terms or entire documents that are not useful for effective retrieval, or whether we could first process those documents that are more likely to be retrieved.

These ideas can be instantiated into different modifications of the inverted index: *static pruning* (Carmel *et al.*, 2001) techniques are offline operations that remove index portions that are unlikely to contain relevant documents. Some static pruning techniques are based upon the statistical properties of documents – i.e., by relying on a summary of the important aspects of a document, or by removing entire terms from the index that rarely impact upon the effectiveness of retrieval (e.g., stopword removal and Latent Semantic Indexing (Deerwester *et al.*, 1990)). Static pruning can be considered as a form of *lossy* compression of the index (in contrast to the *lossless* compression discussed in Section 2.1.2), since the results are no longer safe. Static pruning techniques can be used to develop *tiers* of indexes – smaller, more highly pruned index tiers are more likely to contain the relevant documents for popular queries, while more difficult or less popular queries will need to resort to a larger index tier (Risvik *et al.*, 2003). Tiering and static index pruning are common architectural optimisations in distributed query processing systems, which are employed to reduce the query workloads of these systems as well as the corresponding query processing times. A comprehensive discussion of their implementations and merits can be found in Cambazoglu and Baeza-Yates, 2015, Ch. 4.

In contrast to static pruning, *impact ordering* describes the situation where the documents in the posting lists are sorted not according to their identifier, but according to some measure of their contributions to the relevance of a document to the users' queries. As such, the most "contributing" documents will appear first when processing a posting list, and hence scoring can be terminated early, without reaching the end of the constituent terms' posting lists, while ensuring that the retrieved documents are useful. Indeed, we will describe the special query processing algorithms necessary to deal with these impact-sorted indexes.

Nevertheless, impact-sorting is not a panacea for efficient retrieval, and has notable disadvantages compared to traditional docid ordering. Indeed, the same disadvantages noted in Section 3.5 above for the pre-calculation of term upper bounds (e.g., index updates) also apply for impact-sorting. Nevertheless there are also inherent advantages, particularly concerning the predictability of retrieval times (as postings lists do not need to be entirely traversed). Table 5.1 provides an overview of the advantages and disadvantages of different index orderings – the exact choice appropriate for a given search engine will depend on other factors, such as the necessity of boolean or phrase operators (e.g., to support proximity search). Note that as of 2009, Dean (2009) stated that the Google search engine was using docid-sorting at that time. However, it is likely that other commercial search engines are using impact-sorting, or hybrid combinations of impact and docid-sorting.

In the following, Section 5.1 describes the revised inverted index data structures necessary for impact-sorting, while Section 5.2 describes *score-at-a-time* dynamic pruning optimisations designed to operate directly on an impact-sorted inverted index.

Table 5.1: Pros and cons of docid-sorting vs. impact-sorting of an inverted index.

Docid-sorting	Impact-sorting
Pros	
• Can do phrase/boolean queries based on unigram posting lists • High compression rate • Can append new documents to posting lists	• Lower mean and variance in response times • Fast to decompress • Early termination of a posting list pass less damaging to effectiveness • Easier to predict response times
Cons	
• Difficult to predict response times • Retrieval requires a pass over the entire posting lists	• Difficult to update index • Possible loss in effectiveness

5.1 Data Structures

During the query evaluation, we want to identify as soon as possible the top scoring documents for a given query. Most of the query processing strategies discussed thus far assume that the postings within the posting lists of the inverted index are sorted by docid. An alternative way to arrange the posting lists is to sort them such that the highest scoring documents appear early. In theory, in docid-sorted lists, the top documents could be found at the end of the posting lists, forcing any safe up to K algorithm to traverse them completely. If an algorithm could find them at the beginning of the posting lists, the dynamic pruning conditions can be enforced very early.

The first alternative organisation of posting lists was proposed by Wong and Lee (1993), based on the TFIDF rule (Salton, 1989). This rule asserts that, for each term t in query q and for each document d in the collection, the similarity score between q and d must increase with the term-document frequency $f_{d,t}$ and must decrease with the document frequency f_t. While processing posting lists in decreasing order of term upper bound, Wong and Lee (1993) organised the posting lists in decreasing term-document frequency values, and processed them by grouping postings in individual disk pages. In this way, the pages are processed in decreasing order of similarity score contributions, until an estimated threshold on the retrieval accuracy is reached and the top K documents scored thus far are then returned (behaving then as a Quit strategy). The authors noted that the processing and storage overheads of this *frequency-sorted* index will be higher than a *docid-sorted* index, but these overheads can have a limited impact for environments where updates are done in batch and are infrequent compared to retrieval activities.

The same index organisation is used by Persin (1994) and Persin *et al.* (1996) to propose a TAAT optimisation that leverages the frequency sorting to select the documents to prune. Firstly, Persin (1994) noted that the decision to stop in the Quit and Continue optimisations by Moffat and Zobel (1996) is based only on the global parameters of the collection. The pruning condition of both algorithms completely rejects whole posting lists rather than separate documents within the lists, making it impossible to have "gradual transition" from processing to rejecting terms. Persin proposed a *document filtering* modification of the TAAT strategy based on two *filtering thresholds* f_{ins} and f_{add}. For each posting in any posting list, the term-document frequency $f_{d,t}$ is compared to these two new thresholds. If $f_{d,t}$ is greater than the *insertion threshold* f_{ins}, the corresponding accumulator is created and initialised, if not present, or its partial score is updated, if already present. Otherwise, if $f_{d,t}$ is greater than the *addition threshold* f_{add}, then the partial score is accumulated only if the corresponding accumulator is already in the candidate set. If the document is not in the candidate set, nothing is done in this case, as well as in the case when the term-document frequency passes neither thresholds.

With docid-sorted indexes, the frequency tests must be done for every posting in every posting lists, while with the frequency-sorted posting lists, we start processing the most important document in each list first, then we can stop inserting new accumulators and eventually stop accumulating the partial scores of the existing accumulators. Although the filtering thresholds must be experimentally tuned, the authors showed a large memory saving in terms of the number of accumulators compared to Lucarella (1988) and Harman and Candela (1990).

One of the most popular and effective similarity models is the *vector space model* (Witten *et al.*, 1999; Zobel and Moffat, 2006; Baeza-Yates and Ribeiro-Neto, 2008; Manning *et al.*, 2008; Croft *et al.*, 2009; Büttcher *et al.*, 2010). The vector space model relies on the TFIDF rule for statistically approximate similarity scores between a query and a set of documents.

In such a model, the similarity function in Equation (2.1) can be efficiently represented as follows:

$$\textsc{Score}_q(d) = \sum_{t \in q} s_t(q, d) = \frac{1}{W_d} \sum_{t \in q} w_t(q) \cdot w_t(d) \tag{5.1}$$

Anh *et al.* (2001) introduced the definitions of (i) the *impact* of term t in document d for the quantity $w_t(d)/W_d$ (or *document impact*) and (ii) the *impact* of term t in query q for $w_t(q)$ (or *query impact*). Building upon the ideas of frequency-sorted indexes, the authors proposed to facilitate effective query pruning by sorting the posting lists in decreasing order of (document) impact, i.e., to process queries on an *impact-sorted index*. In such indexes, postings do not store the term-document frequency $f_{d,t}$ anymore, but the document impact $w_t(d)$. However, while term-document frequencies are natural numbers, document impacts are floating point values. Hence, they are not as amenable for compression. Leveraging a technique introduced by Moffat *et al.* (1994) applied to document weights, Anh *et al.* (2001) quantised the impact values, approximating them by b-bit integers, and storing such integers instead of the original real values in the postings. Two successful approaches were proposed to assign impacts to weights, LeftGeom and Uniform. A third one, RightGeom, did not provide a good effectiveness compared to the other two. In the LeftGeom quantisation, a weight w in the real range $[L, U]$ is assigned to one of the 2^b buckets according to the following mapping:

$$w \mapsto \left\lfloor 2^b \frac{\log(w/L)}{\log(U/L) + \epsilon} \right\rfloor \tag{5.2}$$

while in the Uniform quantisation, the mapping is as follows:

$$w \mapsto \left\lfloor 2^b \frac{w - L}{U - L + \epsilon} \right\rfloor \tag{5.3}$$

where ϵ is a small positive value to ensure that the weight U maps to the impact $2^b - 1$ rather than 2^b.

The LeftGeom quantisation has this name since it provides more accurate approximations for values close to the weight lower bound L, while the Uniform quantisation equally divides the $U - L$ range into 2^b buckets (the RightGeom quantisation provides good approximations of large values, close to the weight upper bound U). Given an impact $i \in \{0, \dots, 2^b - 1\}$ and a direct mapping $w \mapsto f(w) = i$, the corresponding inverse weight is defined as:

$$\frac{1}{2}\Big(f^{-1}(i) + f^{-1}(i+1)\Big) \tag{5.4}$$

i.e., the middle value of the range of weights corresponding to that bucket. The experiments conducted by Anh *et al.* (2001) showed that 5 bits are reasonable to encode the impacts with minimal effectiveness losses for both quantisation schemes (with 10 bits, no effectiveness losses are reported) but this value should be tuned when using different collections and/or different similarity functions. Later, Crane *et al.* (2013) confirmed that 5 to 8 bits are enough for small-medium document collections, but for larger collections, from 8 up to 25 bits are necessary, depending on the effectiveness measure. They proposed a simple empirical formula to compute the number of bits b according to the number of documents $|D|$, as follows:

$$b = \left\lceil g + h\sqrt{|D|/10^8} \right\rceil, \tag{5.5}$$

where g and h are linear fitting parameters depending on the used effectiveness measure ($g = h = 5.4$ for MAP, $g = 2.9, h = 54.3$ for P@20).

5.2 Query Processing

During query processing, early termination can be checked every time the impact of postings changes during the posting lists traversal. Posting lists are divided into document impact blocks of postings and interleaved as in Figure 3.5. Then, given a query, the document impacts of a given posting list are multiplied by the query impact of the corresponding term, and the impact blocks are processed in decreasing order according to these values. This query processing strategy on impact-sorted indexes is often called *score-at-a-time* (SAAT), and illustrated in Algorithm 5.1.

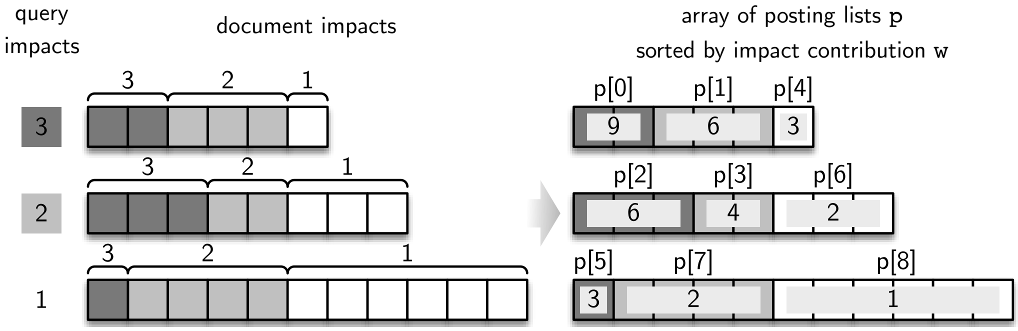


Figure 5.1: How the SAAT algorithm sorts the posting list blocks by impact

The adoption of integer values to encode the impacts and to enforce dynamic pruning allows the accumulators to be integers, and hence efficient techniques can be leveraged to sort them, resulting in a 2× speedup compared to frequency-sorted indexes.

Further investigations on impacts have been discussed in (Anh and Moffat, 2002; Anh and Moffat, 2005b; Anh and Moffat, 2006b; Anh and Moffat, 2006a). In particular, in (Anh and Moffat, 2002), the authors found evidence that the commonly used similarity scores based on the TFIDF rule overstate the role of high impacts and dampens the effect of

Algorithm 5.1: The SAAT algorithm

```
Input  : An array p of $N \times 2^b$ posting lists, one per query term per impact,
         sorted in decreasing order of impact contribution (see Figure 5.1)
         An array w of $N \times 2^b$ impact contributions, one per query term per
         impact, sorted in decreasing order
Output : A priority queue q of (at most) the top $K$ ⟨docid, impact⟩ pairs,
         in decreasing order of impact
ScoreAtATime(p):
    A ← an accumulators map from docids to impact,
        all entries initialised to 0
    for i ← 0 to $N \times 2^b - 1$ do
        current ← p[i].docid()
        while current $\neq \perp$ do
            A[current] ← A[current] + w[i]
            p[i].next()
            current ← p[i].docid()
    q ← a priority queue of (at most) $K$ ⟨docid, impact⟩ pairs,
        sorted in decreasing order of impact
    foreach ⟨docid, impact⟩ in A do
        q.push(⟨docid, impact⟩)
    return q
```

low-impact terms. This is particularly detrimental for the effectiveness of short queries. The authors hence proposed a global *impact normalisation* procedure, to allocate large relative increases to small values and vice versa. Then the new impact values are again quantised, to be represented as integers, and thresholded, to statically remove from the index the lowest-impact blocks of postings. Among the different solutions proposed by the authors, the most successful one consists in reducing by a fixed amount the impact score contribution to any accumulator. This amount is initially zero, but it is increased every time an impact block or a whole posting list is completely processed. In doing so, the score contribution of low impact blocks and query terms is gradually reduced, partially reversing the effects of the normalisation procedure. With all parameters tuned and with the right data structures, this impact normalisation procedure increased both the effectiveness and efficiency of the original impact-sorted indexes.

Anh and Moffat (2005b) proposed a different mechanism to produce normalised impacts. Instead of relying on *quantitative* impacts derived from a pre-determined similarity function, they introduced the concept of *qualitative* impacts. Such impacts are defined locally to each document and then transformed as in (Anh and Moffat, 2002). However, instead of adapting the mappings of Equation (5.2) and (5.3) from a global view (i.e., using collection-level bounds L and U) to a local view (i.e., using document-level bounds L_d and U_d), they proposed to use the ranking position of terms in documents. Hence, for a given document, its terms are ranked according to some criteria, then they are grouped into 2^b buckets, and *rank impacts* assigned to buckets. The experiments reported improvements in effectiveness in terms of mean average precision with BM25 and language models, coupled with the efficiency benefits of impact-sorted indexes already discussed above.

Anh *et al.* (2001) proposed a revised management of the accumulators array. They proposed to keep track of the accumulators through their *values*, by using an array of lists of accumulators indexed by accumulator value. Then they keep track of the top accumulator's value and the lowest K accumulator's value through a set of pointers, that are updated at runtime to ensure that the top K accumulators are correctly identified at the end of query processing.

Anh and Moffat (2006a) proposed a dynamic pruning optimisation approach for SAAT, based on the Continue strategies presented in Section 3.2. The impact blocks are initially processed in an *OR mode* (as presented in Algorithm 5.1). The processing continues in an *AND mode*, i.e., no new accumulators are created, once no document with an existing accumulator can be a member of the final candidate set. At a certain point, we will be able to exactly identify the top K accumulators, so the algorithm proceeds to a third phase in a REFINE-mode, where only these top K accumulators are retained, and correctly scored. The experiments showed that this optimisation approach is able to reduce the memory footprint by 98% w.r.t. SAAT, with a speedup of 1.75×. Note that the proposed optimisation is safe up to K compared to SAAT. The authors also discussed an unsafe modification to the proposed algorithm. After the initial *OR mode* processing, a *fidelity control knob* Q controls the percentage of postings to process after the first phase. They found that processing $Q = 30\%$ of the remaining postings results in a good retrieval performance, for both long and short queries. This approach is discussed also by Lin and Trotman (2015) and implemented in the JASS open-source search engine. The authors applied linear regression to correlate a time deadline (on the query processing time) with the maximum number of postings to process to meet the deadline ρ (e.g., 10^3, 10^4 or 10^5 postings). Once this number of postings ρ is processed, no new impact blocks are processed. Mackenzie *et al.* (2017) further investigated this approximate early termination approach, a.k.a. *anytime ranking*. Instead of terminating the processing after a *fixed* amount of postings is processed (10% of the documents in the collection as empirically observed by Lin and Trotman (2015)), they proposed to stop after processing a given *percentage* of the total postings in the query terms' posting lists, on a per-query basis. According to their experiments, the fixed threshold may result in reduced effectiveness as the number of query terms increases, but conversely it gives a very strict control over the tail latency of the queries.

Strohman and Croft (2007) further refined the memory-optimised safe up to K approach described in (Anh and Moffat, 2006a), by reducing the number of accumulators gradually during the *AND mode* phase. The authors proposed to *trim* in a MaxScore style the accumulators that cannot beat the current threshold during the second phase, and optimised the accumulators data structure to leverage the cache, together with a new skipping scheme, obtaining a 1.69× speedup compared to the Anh and Moffat (2006a) strategy.

Jia *et al.* (2010) introduced several improvements to the SAAT algorithm. They proposed to manage the priority queue `q` during the posting list traversal. Moreover, they showed that in some cases the accumulator initialisation can be quite expensive, i.e., when the `memset` library function is used. They discussed a management of the accumulators array similar to the *paging* mechanism in the operating system's virtual memory. Lin and Trotman (2017) discussed how to include early termination in SAAT. Every term has an associated upper bound, that is updated, i.e., decrease, every time a corresponding block impact list is completely processed. By storing (and updating) the sum of the term upper bounds during query processing, we can stop creating *new* accumulators when the top K accumulator scored thus far is larger than the sum of upper bounds. We can also stop updating accumulators that are not in the priority queue when the difference between the top K and the top $K+1$ accumulators is larger than the current sum of upper bounds, since no existing accumulator can further accumulate enough impact to beat the current top K accumulators.

Trotman (2014) investigated the performance of integer compression algorithms for frequency-sorted and impact-sorted indexes. The author demonstrated some space inefficiencies in existing SIMD-based codecs, in particular with short posting lists, and proposed a new SIMD compressor (QMX), that is more time and space efficient than the SIMD codecs. Trotman and Lin (2016) provided further experiments for QMX. Lin and Trotman (2017) found that the best performance in query processing with SAAT is indeed obtained when *no compression* is used, even if the advantage w.r.t. QMX is small (∼5%). Moreover, uncompressed impact-sorted indexes can be up to two times larger than their QMX-compressed versions.

Finally, we note that Crane *et al.* (2017) performed a detailed investigation of the performance of rank-safe SAAT with respect to WAND and BMW. They found that, for $K = \{10, 100, 1000\}$, SAAT performs worse than both WAND and BMW, even if the average response time of SAAT does not exhibit a dependency on K. A multi-threaded implementation of SAAT is further discussed in (Mackenzie *et al.*, 2017).

5.3 Summary

This chapter introduced offline modifications to the content and structure of an inverted index to prioritise the processing of documents that are more likely to be retrieved. We focused upon impact-sorted indexes, where significant efficiency improvements can be obtained with negligible negative impacts on effectiveness. It is likely that current modern Web search engines adopting a combination of dynamic pruning with static pruning and impact-sorting strategies.

In the next chapter, we investigate the role of the techniques discussed in Chapters 3 and 5 in cascading search architectures. We also provide insights into the efficient deployments of learning-to-rank infrastructures, which can benefit the search engine's effectiveness by re-ranking a set of K documents.

6 Learning-to-Rank & Cascades

Over the years, many different ranking models have been proposed in order to score documents in response to a query. Such models strongly depend on the sources of information available to characterise the documents. In the early years of Information Retrieval (IR), the main source of relevance was the presence of the query terms in a document. Next, the relevance of a document to a query was better formalised using the Vector Space Model (Salton *et al.*, 1975) or using the Probabilistic Ranking Principle (Maron and Kuhns, 1960). These bag-of-words approaches leverage the occurrences of query terms in the whole collection and in each single document, modeling documents and queries as vectors of term/document frequencies.

In the past there has been significant evidence that IR can benefit from data fusion, also called metasearch, whereby the outputs of multiple and different retrieval systems are combined into a single ranking of documents. Various data fusion techniques, such as CombSUM and BordaFuse, were proposed to build an unsupervised combination of IR system rankings, for instance by combining raw retrieval scores (Bartell *et al.*, 1994; Fox and Shaw, 1994), normalised retrieval scores (Montague and Aslam, 2001; Manmatha *et al.*, 2001) or simply leveraging the ranks of retrieved documents (Montague and Aslam, 2002). Data fusion benefits effectiveness through a number of manners, such as the *chorus effect*, whereby if multiple constituent systems agree on the estimated relevance of a given document, then this document is more likely to be relevant (Vogt and Cottrell, 1998). The perceived benefits of data fusion led to the development of various Web metasearch engines for some time, such as HotBot and Dogpile. Later, supervised data fusion techniques were developed (e.g., (Lillis *et al.*, 2006)).

However, the clear disadvantage of data fusion is that multiple IR systems need to be queried to attain the perceived effectiveness benefits. Hence, it could be argued that it would make more sense to combine the attributes of multiple systems within a single system, for both efficiency purposes and to eliminate the need for license payments.

On the other hand, with the Web, new sources of information about the documents have been increasingly identified. Measures such as the importance of a Web page (e.g., PageRank, number of inlinks/outlinks), additional document statistics (e.g., term frequencies in the title or body fields, anchors text, term proximity) and search engine interactions (e.g., URL clicks) can be exploited as relevance signals. Collaborative and social platforms such as Wikipedia, Twitter and Facebook are exceptional sources of relevance signals. For example, Wikipedia titles can be used for entity annotation in queries, while social media can capture the users' behaviour, and identify fresh pages that are relevant to new or trending queries.

With such an abundance of signals to be taken into account when modeling the relevance of documents w.r.t. queries, simple ranking functions such as those corresponding to Equation (2.1) show their limits. Even with a simple linear combination of signals, the weighting parameters of the linear function still need to be adequately estimated. Therefore, to address the challenge of effectively combining large numbers of relevance signals, machine learning approaches have proven successful in producing effective relevance ranking algorithms, generally referred to as *learning-to-rank* (LTR). In this monograph, we are not focusing on specific LTR methods and learning-based retrieval algorithms. Instead, we are rather interested in their implementation and usage in query processing. For a comprehensive survey of LTR methods and approaches, and for additional details on the learning-to-rank paradigm, please refer to the seminal monograph of (Liu, 2009).

One important advance in search engine architectures facilitated by learning-to-rank is the separation of the ranking process into cascades. In particular, bag-of-words retrieval models such as BM25 are sufficiently fast that they can be efficiently calculated when querying over all documents matching any of the query terms, even when the underlying index contains millions of documents. Such retrieval models use very few statistics (e.g., term frequency, document frequency, document length) that are efficiently stored and accessed in the inverted index, and are easily combined in a predefined manner. In contrast, LTR models combine hundreds of arbitrary features, and they can be combined in the most effective way thanks to training data (Dang *et al.*, 2013). Due to the computational cost of extracting or computing hundreds of features for every query-document pair, the LTR models are not applied directly on the document collection, i.e., to rank all documents matching a query. Instead, they are deployed in a pipelined fashion by conducting first a simpler (base) preliminary ranking stage, before one or more subsequent more expensive ranking processes are applied, in a cascading manner.

Figure 6.1 provides an overview of how a search engine operates in a basic cascading manner. During query processing, the base ranking stage uses simple ranking functions to retrieve from the whole document collection a *sample* (or candidate set) of documents of sufficient recall effectiveness. Approaches from Chapters 3 & 5, such as static and dynamic pruning, are easily deployable at this stage. In the second stage, further computationally-expensive and highly discriminative features for the documents in the sample are generated or extracted from other sources, and the LTR ranking stage reranks the sample's documents, focusing on high precision results at the top positions. The final top K documents of the reranked list are then returned to the user.

This separation into stages also has important benefits for faster training, in that documents can be ranked once, extra features calculated, and the learning process conducted 'offline' on a single machine, disconnected from the underlying retrieval engine. Various learning-to-rank datasets, including the LETOR ones released by Microsoft Research[1] were created in this manner, allowing researchers to evaluate learning-to-rank techniques on shared datasets.

More complex cascading architectures have been devised. For example, Yin *et al.* (2016) described the Yahoo search engine distributed architecture being composed of a preliminary *recall* stage performing boolean retrieval over the document collection, followed by a *first round* stage performing lightweight scoring. Next, there is a *core ranking function* stage, based on LTR models trained simply on query-document pairs. These stages run on the *index serving nodes* of their distributed infrastructure. A single *blending node* is responsible for merging and sorting the top results from multiple index serving nodes, which implements a *contextual reranking* stage, based on LTR models trained with information extracted from other candidate results for the same query, to compute the final top results to return. A similar architecture is described by Risvik *et al.* (2013), detailing part of the search infrastructure of Microsoft's Bing.

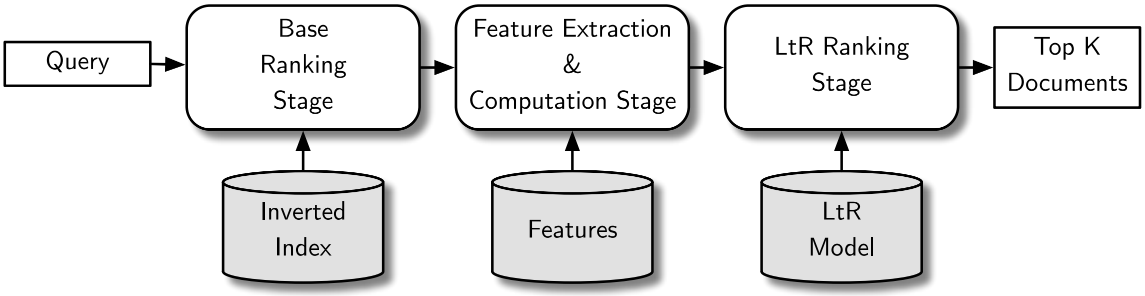


Figure 6.1: Basic stages of a cascading search engine

In general, the efficient application of learning-to-rank within search engines, while deemed useful, has seen correspondingly less research compared to the generation of effective learned models. In fact, the efficiency of LTR model applications mainly depends on three factors: (1) the efficiency of the initial base ranking stage, (2) the calculation/extraction of the required features, and (3) the efficient application of the learned models to combine those features into a final improved ranking. These factors also depend on the sample size, i.e., the number of documents retrieved in the base ranking stage.

As a consequence, in the rest of this chapter, we will address the candidate generation for learning-to-rank based on the initial base ranking stage (Section 6.1), the calculation of features for learning-to-rank (Section 6.2), and the efficient application of learned models (Section 6.3).

[1] https://www.microsoft.com/en-us/research/project/letor-learning-rank-information-retrieval/

6.1 Candidate Generation for Learning-to-Rank

In a search engine using learning-to-rank, the base ranking stage needs simply to identify a set of documents of sufficient recall effectiveness (i.e., the sample), in an as efficient manner as possible. Clearly this can be achieved through a direct application of the dynamic pruning techniques described in Chapter 3, and/or those in Chapter 5. This allows for two avenues that benefit efficiency, namely (i) retrieving less documents in the base ranking stage, i.e., a smaller sample size (which enhances efficiency) and/or (ii) sacrificing the safety/effectiveness of the dynamic pruning strategies.

Macdonald *et al.* (2012c) examined the second avenue, the dynamic pruning safety, by varying the F aggressiveness parameter of WAND (see page 51). When re-ranking 1000 documents, there was no change at rank 20 in the effectiveness of the sample for $1 \leq F \leq 3$, but when re-ranked by a LambdaMART learning-to-rank model, the LambdaMART performance was much less stable for different F values. As $F \rightarrow 10$, the effectiveness of the learned model decreased, with corresponding efficiency gains. Of particular note, the effectiveness was even observed to significantly improve for $F = 1.75$, despite failing to retrieve 82 relevant documents across the 50 queries. This is caused by a bias inherent in unsafe WAND, where retrieval becomes more focused on lower numbered docids in the posting lists. Indeed, by artificially increasing the threshold τ by the factor F, the threshold for unsafe WAND causes more documents to be prevented from entering the top K documents. Early in the traversal of the posting lists, when τ is lower, documents can still enter into the retrieved set. However as τ becomes higher, more pruning occurs, even for documents that would have made the retrieved set for $F = 1$. This explains unsafe WAND's bias towards lower docid documents.

While Liu (2009) discussed the use of 1000 documents as a candidate set, this came without empirical justification. Indeed, some other learning-to-rank approaches in the literature used smaller candidate sets, in the order of 10s of documents (which may constitute an easier problem for learning an effective ranking model). Macdonald *et al.* (2013a) studied the effectiveness of a variety of learning-to-rank techniques as the sample size is varied. In general, on the large ClueWeb09 B corpora, effectiveness was markedly reduced for adhoc and mixed-task queries if less than 1000-2000 documents were retrieved during the base ranking stage. On the other hand, for easier navigational tasks, less documents were needed, particularly if the anchor text of the documents was included in the document representation.

A growing body of work has encompassed varying the initial number of documents necessary to rank on a per-query basis. The first of these, selective pruning (i.e., the selective adjustment of dynamic pruning techniques on a per-query basis), initially proposed by Tonellotto *et al.* (2013), is based on query efficiency techniques, and was described earlier in Section 4.3.1. More recent approaches have addressed the problem by *learning* the number of documents to retrieve in the candidate set (Culpepper *et al.*, 2016). However, a recurring problem is the availability of sufficient queries with relevance labels to allow a successful learning. This challenge has been overcome by the use of *reference lists* (Clarke *et al.*, 2016; Culpepper *et al.*, 2016), which, given a standard evaluation metric M,[2] allow for comparative effectiveness measurements between a ranking assumed to be effective and another ranking, using a measure called Maximised Effectiveness Difference (MED-M). For instance, Mackenzie *et al.* (2018) learned to predict the configuration of the initial candidate set generation – either the minimum sample size suitable for a given query obtained from BMW, or the aggressiveness parameter ρ of the JASS SAAT technique (see Section 5.2). In doing so, their objective is to minimise changes in the MED-RBP measure compared to the reference ranking lists produced by a reference system. In using their predictions, they adopted a hybrid approach, where for each query, a decision is made whether the query should be addressed by a BMW dynamic pruning strategy or a JASS approach, to minimise tail latency, and how the selected retrieval technique should be configured in terms of aggressiveness. Their predictions demonstrate no loss in effectiveness compared to their learned model approach, as evaluated on the TREC 2009 Web track topics.

On the other hand, not all proposed techniques in the literature for the first stage of retrieval actually perform ranking. Asadi and Lin (2012) described a technique using Bloom filters. Indeed, they noted that in scenarios such as Web search, many queries are processed conjunctively, i.e., only documents that contain all the query terms are considered. This being the case, the authors proposed that the postings list be stored both as a compressed sequence of integers and as a Bloom filter – a fast and small data structure that supports $O(1)$ approximate set membership tests. In this case, the Bloom filter permits to quickly determine whether a docid occurs in a posting list (with a given accuracy). This allows the intersection to be quickly made. Building on this, Goodwin *et al.* (2017) described a signature-based approach for representing the candidate set generation phase called BitFunnel. BitFunnel has since been deployed into Microsoft's Bing search engine. We discuss it further in Section 7.2.

Once the docids for the candidate set have been identified, the next step is to calculate the additional signals (or features) for every document, such that the learned model has a further chance to identify the most relevant documents to place at the top of the re-ranked candidate set. In the next section, we further discuss the efficient calculation of such features.

6.2 Feature Calculation in Learning-to-Rank

The basic model for LTR is as follows. Given a specific query q, a document d is represented by a vector of *features*. Each feature relates to one or more relevance signals of the query-

[2] Such as MAP, NDCG or Rank Biased Precision (RBP)

document pair (q, d). A numeric *relevance judgement* is associated to each query-document pair. A machine learning algorithm is then trained to produce a *learned model* combining all input features. Figure 6.2 shows the features for three documents, as used in the LETOR datasets. Typically, the learned model is trained to reproduce as accurately as possible the relevance judgements of a training set of query-document pairs by minimising an effectiveness-specific loss function (Liu, 2009). At query processing time, the learned model is used to rank documents for new user queries, by computing the same input features for each query-document pair and producing a ranking of the documents.

```
2 qid:1 1:1    2:1.9 3:0.4 # docid=5
0 qid:1 1:0.99 2:0.2 3:0.5 # docid=941
1 qid:1 1:0.88 2:0.5 3:0.8 # docid=83
```

Figure 6.2: An example of a LETOR-formatted learning-to-rank training file.

There are various classes of features implemented within a learning-to-rank deployment. Many of these were historically included in the LETOR datasets (Qin *et al.*, 2010), and encapsulated various research techniques available at that time. The common classes are:

- *Query-dependent* features: These are document features that vary according to the query. Typically, these will encapsulate various term weighting models, including those calculated on separate fields or proximity/dependence models. Macdonald *et al.* (2013b) showed that including multiple such query-dependent features within a learning-to-rank model significantly benefited effectiveness.
  More recently, neural semantic matching models have been proposed, which use dense vector representations of terms and deep learning to achieve a more accurate similarity between queries and documents. While their integration into search engine architectures has not yet seen a wide examination in the literature, approaches such as DUET (Mitra *et al.*, 2017), DeepRank (Pang *et al.*, 2017), and the weakly supervised RankProb model (Dehghani *et al.*, 2017) were applied as re-rankings of candidate document sets created by, for instance, BM25. For this reason, we describe features based upon such neural semantic similarity models as query-dependent features (they need both the query and the document). However, note that these features require access to some representative content of the documents, as might be found in the direct (forward) index.
- *Query-independent* features: These are document features that have the same value for each query. The ubiquitous PageRank and various other link-analysis features fall into this category, as well as URL length, spam features and other content-based document quality indicators – Bendersky *et al.* (2011) described a number of such features, including fraction of words in a table, fraction of stopwords covered by a document, to name but a few.

- *Query-only* features: Such features are document-independent, in that they have the same value for each document (Macdonald *et al.*, 2012b). Query analysis, such as query type identification and query performance predictors, falls into this category. These features permit the adaptation of the ranking strategy for different query categories such as query type (informational vs. navigational, easy vs. difficult, presence of an entity, or news-related, etc.).
- *Contextual* features: These features take into account contextual information of existing features from other candidate results for the same query. Lucchese *et al.* (2015a) proposed *rank-based* features, to provide additional information about some ordering properties of a document compared with the other candidate documents. For example, a new feature corresponding to the rank of a document after sorting the candidates w.r.t. a given feature can capture relatively better or relatively worse concepts over the current candidate set. Yin *et al.* (2016) proposed new contextual features, such as the mean/variance of specific feature values in the candidate set, and so on.

The manner in which the features are calculated is of particular interest. For instance, query-independent features may simply be recorded in a fast-access memory data structure (i.e., an array of values for each document held in memory – these may be floating point values). However, some tree-based learning techniques can make use of quantised feature values, making use of distributions learned on the training data. Hence, these techniques require fewer bits to record the feature values for a given document (Li *et al.*, 2007).

An alternative architecture for storing the values of the query-independent features is to record them in the posting lists. This makes each of the postings larger, and repeats the feature values for every posting in every document. This causes a large memory overhead, and for this reason, this architecture is rarely used. On the other hand, feature values are immediately available during scoring without requiring a random access lookup.

On the other hand, calculating additional query-dependent features, such as those based upon proximity or field information, requires access to the posting information, normally recorded in the inverted index. This presents a challenge: the learning-to-rank paradigm suggests that the calculation of additional query-dependent features should only occur once the initial set of the top K documents to be re-ranked has been identified, as part of a first phase retrieval (as described in Chapters 2-5). Indeed, it is considered too expensive to calculate the additional features for all documents that contain one or more query terms and that might make the top K documents. However, the calculation of additional query-dependent features requires access to the postings in the inverted index, which are not readily available for random-access.

There are a few solutions for this challenge, which have advantages or disadvantages in terms of flexibility and efficiency:

- *Inverted Index Re-traversal:* In this architecture, which is notably inefficient, the relevant inverted index posting lists are re-traversed for the documents in the final top K set. Figure 6.3 illustrates the flow of data for this architecture. This can be reasonably efficient since the set of documents being scored is small, and skipping (Section 2.1.2) may be used to skip more quickly to the correct postings. Yet, the postings for those documents being rescored with additional query-dependent features must be decompressed.[3]

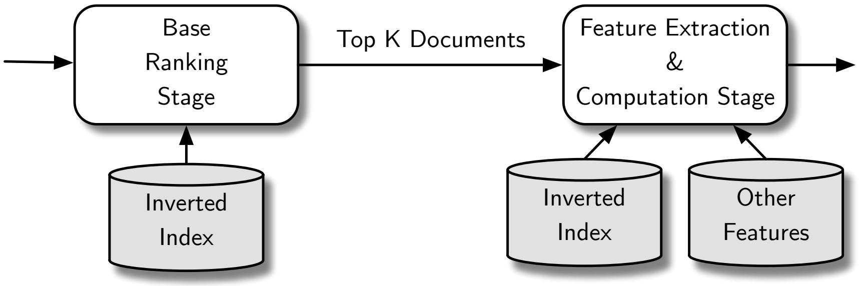


Figure 6.3: Using an inverted index for calculating additional query-dependent features.

- *Caching of the postings:* (Macdonald *et al.*, 2013b) described the FAT architecture implemented by the Terrier IR platform (Macdonald *et al.*, 2012a), whereby the matching postings of the documents that are admitted to the top K heap are cached. At any point of time, at most $K \times n$ postings are kept in the heap, i.e., for every document, one for each query term. Figure 6.4 illustrates the flow of data for this architecture, which is described as ‘fattening’ the result set with postings. This allows the computation of additional query-dependent features without accessing and decompressing the posting lists in the inverted index a second time. However, this architecture is only applicable for computing such additional query-dependent features in the case of query terms that occurred in the original query, since the fattened result set will not contain the postings of any further query terms.
- *Direct Index:* Asadi and Lin (2013) described a representation where the ‘forward’ or ‘direct’ index[4] is used for calculating additional query-dependent features in a document-centric manner. Figure 6.5 illustrates the flow of data for this architecture. They described several implementations of such an index, concentrated around either an array of term ids – one for each term position – or a sparser representation, similar to that used in an inverted index posting list (termids, frequencies, term positions). In both cases, termids are assigned such that more frequently occurring words obtain lower termids, thereby benefiting compression. In particular, if the term position

[3] At the time of writing, our understanding is that Solr’s LTR implementation is based on index re-traversal, based on conversations with software engineers familiar with such an implementation.

[4] Also called document vectors by Asadi and Lin (2013).

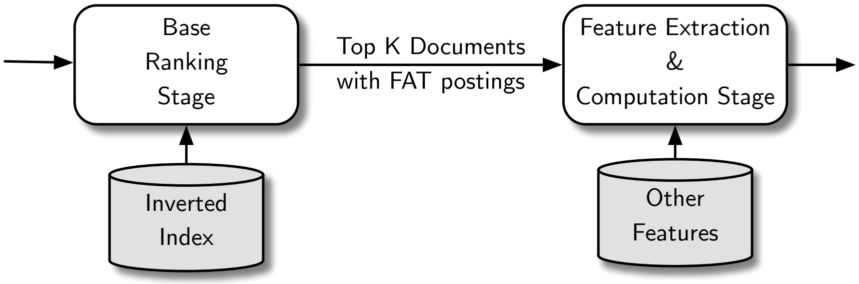


Figure 6.4: Retaining FAT postings from the inverted index for calculating additional query-dependent features.

information is not required for the first pass retrieval, then the inverted index does not need to include the position information, and this can instead be kept solely in the direct index. On the other hand, even with posting compression, a direct index can take significant space over-and-above the inverted index. Indeed, Asadi and Lin (2013) found that the space consumption of a direct index with positional information combined with a non-positional inverted index is similar to that of a positional inverted index (depending on the compression scheme used, 70 - 86 GB vs. 75GB for the 50 million documents of the ClueWeb09 B collection). To address the space issue, they proposed a space-efficient hashing scheme for direct index contents to reduce the number of bits needed to encode the termids of a given document, giving a total index of 58 GB.

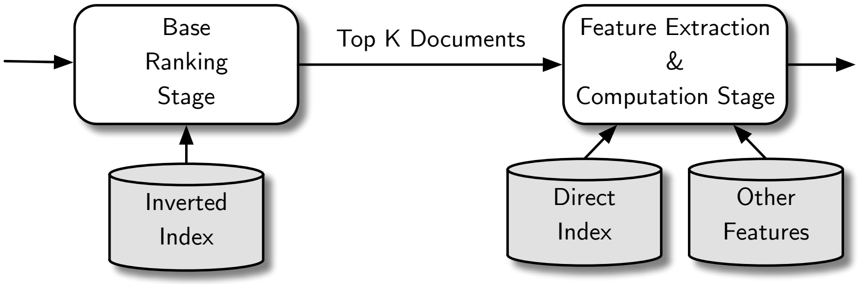


Figure 6.5: Using a direct index for calculating additional query-dependent features.

Compared to Terrier's FAT framework, we note that using the document vectors approach of Asadi and Lin (2013) allows additional query-dependent features to also target further query terms that were not present in the original query, such as those derived from query expansion/rewriting. Such an approach also facilitates the use of deep neural network matching models, such as those of Dehghani *et al.* (2017), which learn to match queries to documents, and hence also require the content of the documents in the candidate set.

We finally highlight the work of Arroyuelo *et al.* (2012), which also considered whether positional information needs to be indexed. In particular, they demonstrated that the compressed representation of the raw text of a document can be achieved in a space only 12% larger than the positional index – such an index data structure could therefore have uses for both proximity feature generation and snippet generation.

6.3 Application of Learning-to-Rank Models

Following the taxonomy introduced by Capannini *et al.* (2016), the LTR models typically adopted in IR systems or Web search engines fall into the following three classes, ordered by increasing complexity:

1. *linear* models, where a scalar product between the input features and the learned weights is computed;
2. *neural network* models, where neural networks are trained to minimise specific loss functions;
3. *forests of trees* models, where thousands of regression trees are used as ranking models.

This classification groups learning-to-rank algorithms by their implementation and complexity at the query processing stages. The main learning algorithms producing linear models are Coordinate Ascent (Metzler and Croft, 2007) and SVM-Rank (Joachims, 2002). The computations performed during query processing by such models are extremely cheap and fast, and modern CPUs can easily parallelise the required operations. The overall cost is that of a linear combination of the number of input features.

Neural network models employed in LTR algorithms such as RankNet (Burges *et al.*, 2005), ListNet (Cao *et al.*, 2007) and SortNet (Rigutini *et al.*, 2011) have a computational cost, which depends on the number of input features and their internal complexity, i.e., the number of nodes in the inner layers. The computations they need to perform and the non-linear activation functions they employ (e.g., sigmoid and hyperbolic functions) make the application of such models more complex and more time-consuming than linear models. Their overall cost is that of a linear combination of the number of input features, i.e., the size of the input layer, and the number of hidden nodes, i.e., the size of the hidden layer.

The forests of trees class of models includes all LTR algorithms that produce a ranking model based on decision trees. Such trees take boolean decisions at each internal node, comparing an input feature value with a given threshold and traversing the tree depending on the outcome of such decisions. When an exit node is reached, a score value is produced. A large number of learning algorithms fall in this class, depending on the structure of trees and the aggregation method of the trees' outputs. RankBoost (Freund *et al.*, 2003) employs one-level decision trees (a.k.a. decision stumps) whose outputs are linearly weighted to score documents. Random forests (RFs) (Breiman, 2001) use multi-level regression trees to independently predict document scores whose arithmetic mean is used as the predicted score of each document. Moreover, gradient-boosted regression trees (GBRTs) (Friedman, 2001) use multi-level regression trees, whose outputs are linearly weighted to compute the final scores. GBRT rankers outperform all other classes of rankers in terms of the quality of their results, while their complexity mainly depends on the number of trees employed in the model and the number of their exit nodes. A detailed analysis of the time complexity and the effectiveness-efficiency tradeoff of such algorithms can be found in (Capannini *et al.*, 2016). Despite their complexity, these models are successfully adopted in several industrial scenarios, not just related to query processing. For example, Facebook used boosted decision trees to transform input features concatenated with a sparse linear classifier in their click prediction systems for online advertising (He *et al.*, 2014) and for a variety of internal applications (Hazelwood *et al.*, 2018). Similarly, Microsoft boosted neural networks with GBRTs for click-through prediction in sponsored ads (Ling *et al.*, 2017). Yandex also adopted a similar approach (Trofimov *et al.*, 2012). Microsoft's Bing search engine[5], Amazon Search (Sorokina and Cantú-Paz, 2016) and Yahoo (Yin *et al.*, 2016) have all employed gradient-boosted trees for various ranking problems.

Given the vast success and wide adoption of forests of regression trees, in the following we present the main approaches used for their efficient implementation in the query processing stages (Section 6.3.1), as well as some structural LTR optimisations such as early termination strategies (Section 6.3.2).

6.3.1 Query Processing with Forests of Regression Trees

Let $\mathcal{F} = \{f_1, \ldots, f_m\}$ denote the ids of features in a given LTR setting. The vector $\mathbf{x} \in \mathbb{R}^m$ represents the relevance signals, or feature values, of a given document for a particular user's query that is being processed. A ranking model based on an additive ensemble of regression trees is composed by a set of trees $\mathcal{T} = \{T_1, \ldots, T_n\}$, or a forest. Every tree T_i receives the feature vector as input, and produces as output a real-valued number s_i representing the tree's score contribution to the final score of the document. The score contributions of the trees for the same document are combined into a final document score. Documents are sorted in decreasing final score, and the top K highest-scoring documents are returned as the results list to the user submitting the query or, alternatively, to the next query processing stage in the cascade. Every tree T_i in the forest is composed of a set of binary branching nodes $\mathcal{N}_i$ and a set of leaf nodes $\mathcal{L}_i$. Each branching node performs a boolean test on the value $\mathbf{x}[f_i]$ of a specific input feature $f_i \in \mathcal{F}$ and a constant threshold value γ, i.e., the nodes check if the condition $\mathbf{x}[f_i] \leq \gamma$ is true or false. If the condition is true, the *left* branch is followed, otherwise the *right* branch is followed. The traversal of

[5] https://www.microsoft.com/en-us/research/blog/ranknet-a-ranking-retrospective/

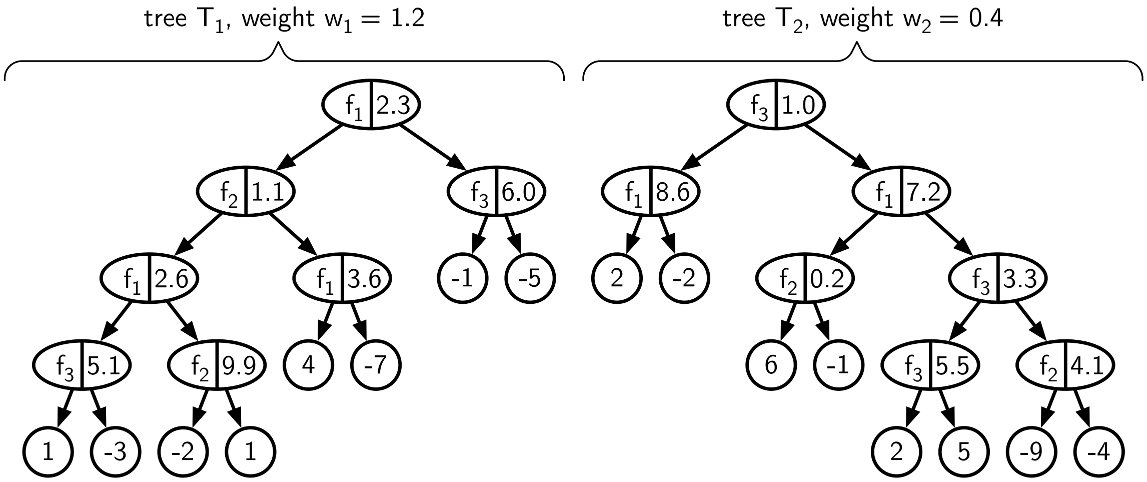


Figure 6.6: A forest of two decision trees with three features. Branching nodes contain the feature id (left) and the threshold value (right), while the leaf nodes denote the score contributions.

the tree continues until a leaf node is reached, and a score contribution $s_i(\mathbf{x})$ is returned. This leaf node is called the *exit leaf* and is denoted by $e_i(\mathbf{x})$. The tree traversal process is repeated for each tree in the forest, and the final score $s(\mathbf{x})$ is finally computed as a weighted sum of score contributions:

$$s(\mathbf{x}) = \sum_{i=1}^{n} w_i s_i(\mathbf{x}). \tag{6.1}$$

An example of a forest of two decision trees with three features, including tree weights, threshold values and score contributions, is depicted in Figure 6.6. Table 6.1 summarises

Table 6.1: Dimensions in a GBRT query processing stage (Chapelle and Chang, 2011).

Number of trees per stage	1,000–20,000
Number of leaf nodes per tree	4–64
Number of documents per query	1,000–10,000
Number of features per document	100–1,000

the complexity of the GBRT rankers deployed at the last stage of a typical query processing cascade (Chapelle and Chang, 2011). With such complex models, it is of paramount importance to efficiently exploit the underlying CPU architecture. The following two main CPU components are exploited for the efficient processing of queries in LTR:

- *cache memory hierarchies*, to speed up the access to data and the execution of compiled code;

- *superscalar architectures*, where multiple instructions can be simultaneously executed.

Cache memory is a small but extremely fast memory holding the recently accessed data and instructions. The performance benefits of cache memory depend on the access patterns of the running program, i.e., the sequence of memory locations being read and/or written during its execution: larger amounts of program instructions/data found in the cache (i.e., a cache hit) lead to faster programs. Moreover, a program's data structures and their access patterns should take into account the presence of cache memories, to exploit spatial and temporal localities and consequently reduce the cache misses.

Superscalar architectures are fully exploited when programs include many *independent instructions* that can be executed in any order and thus in parallel. A *data flow dependency* between a pair of consecutive instructions or a *control dependency* caused by a branching instruction prevent the parallel execution of instructions. Modern CPUs exploit *branch predictors*, which guess the branch directions (taken/not-taken) to prefetch and execute the correct next instruction.

A forest of decision trees can be naively implemented using *conditional statements* (`if then else`) or *conditional operators* (`?:`). In both cases, the compiler can generate very efficient code, reorganising internally the conditional statements/operators. The compiled code size is proportional to the number of trees in the forest and to the number of branching nodes. As reported by Asadi *et al.* (2014) and Dato *et al.* (2016), these direct code translation approaches are efficient for ensembles of trees with a small number of features. However, these approaches suffer from two performance bottlenecks. Firstly, when the generated code does not fit into the cache. Secondly, since the next branch to take is known only after the boolean test is evaluated, the next instruction to be executed is not known, causing a large number of control hazards.

Alternatively, a decision tree could be implemented with a simple binary tree with pointers. Every branching node in the tree contains the feature id, the threshold value and the pointer to the left and right children. Every leaf node contains its score contribution. A C++ implementation based on pointers exhibits poor reference locality and many control hazards, introduced by the boolean tests (Hennessy and Patterson, 2011). Asadi *et al.* (2014) proposed the Struct+ algorithm, to manually manage the memory allocation of nodes to tackle the reference locality problem. Dato *et al.* (2016) reported that implementations based on code generation clearly outperform the Struct+ approach. To address the control hazards in Struct+, Asadi *et al.* (2014) proposed to re-organise the computation, transforming the control hazards into data hazards, with an algorithm called Pred (also explored in previous works such as (Sharp, 2008; Van Essen *et al.*, 2012)). Each tree is visited in breadth-first order and stored in an array of nodes. Each node n_i contains the feature id f, the threshold value γ and an array a of two elements pointing to the left and right children nodes. Leaf nodes have self-pointing children indexes and dummy feature id/threshold values. An example is shown in Figure 6.7.

0	f_1	2.3	1	12	5	⊥	⊥	5	5	10	⊥	⊥	10	10
1	f_2	1.1	2	9	6	f_2	9.9	7	8	11	⊥	⊥	11	11
2	f_1	2.6	3	6	7	⊥	⊥	7	7	12	f_3	6.0	13	14
3	f_3	5.1	1	12	8	⊥	⊥	8	8	13	⊥	⊥	13	13
4	⊥	⊥	4	4	9	f_1	3.6	10	11	14	⊥	⊥	14	14

Figure 6.7: Pred data structure of tree T_1 in Figure 6.6.

The output of a boolean test is used as the index of the child node, to retrieve the next node to be processed. The visit of tree T of depth d_T is statically un-rolled into d_T operations, starting from the root node, as follows:

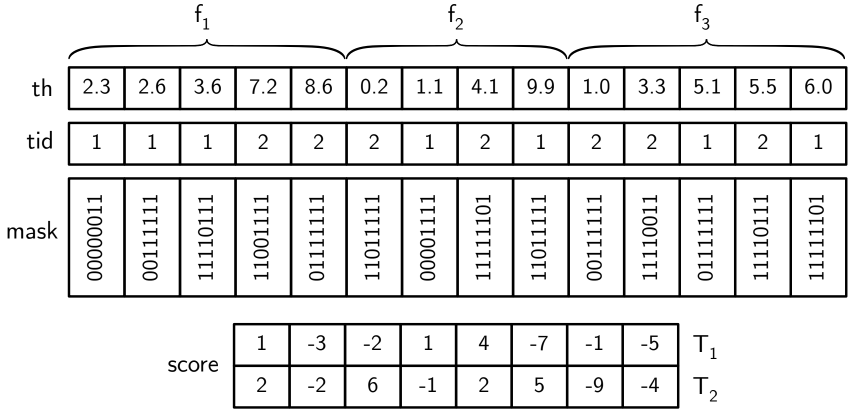
$$i \leftarrow 0$$

$$d_T \text{ times} \begin{cases} i \leftarrow n_i.a\Big[\mathbf{x}[n_i.f] > n_i.\gamma\Big] \\ i \leftarrow n_i.a\Big[\mathbf{x}[n_i.f] > n_i.\gamma\Big] \\ \vdots \qquad\qquad \vdots \\ i \leftarrow n_i.a\Big[\mathbf{x}[n_i.f] > n_i.\gamma\Big] \end{cases}$$

At the end of the visit, the exit leaf is identified by variable i, and a look-up table is used to retrieve the score contribution of the tree. The Pred algorithm successfully transforms control dependencies into data dependencies: the next instruction to be processed is always known. Nevertheless data dependencies are always present, and data locality is not enforced, since the accesses to the array depend on the input feature vector. Asadi *et al.* (2014) then proposed the VPred algorithm, a vectorised version of Pred, which interleaves the scoring of multiple documents. VPred scores multiple documents per tree, allowing the processor to execute instructions in parallel on distinct documents.[6] Their experiments showed that VPred clearly outperforms all other approaches based on static code generation, as well as Pred. Such results are confirmed by experiments performed on different datasets by Lucchese *et al.* (2015b) and Dato *et al.* (2016). Tang *et al.* (2014) improved VPred by laying out the trees data in a cache-conscious way, namely to optimise data traversal for better temporal cache locality. They proposed to partition the t trees of the ranking model and the s documents to process in *blocks* of $p < t$ trees and $q < s$ documents, respectively. The values of p and q are chosen in such a way that the fast cache memory can easily hold the relevant data, and the processing is performed block by block. The authors reported experiments with up to 50% improvements over VPred.

[6] Indeed, recall that LTR is always applied to compute the scores of all documents in a candidate set.

The QuickScorer family of algorithms is the most recent solution for the efficient traversal of the forests of regression trees. The original QuickScorer algorithm is presented in (Lucchese *et al.*, 2015b; Dato *et al.*, 2016). The core idea of QuickScorer is to process the trees in a coordinated way and not one tree at a time, adapting the processing to the CPU characteristics. This is accomplished through computing the final score of a document by identifying the branching nodes whose boolean tests evaluate to false (*false nodes*). In QuickScorer, a tree structure is represented by a set of *mask* bitvectors, one per branching node. Every mask bitvector contains one bit per leaf. All bits are set to 1, but the i-th bit is set to 0 if the corresponding leaf is in the left sub-tree of the branching node. In doing so, we are able to identify the set of unreachable leaves of the tree from a false node. Then QuickScorer groups all branching nodes of the trees by feature id, to efficiently identify the false nodes. For each feature id $f \in \mathcal{F}$, a triple (i, mask, γ) per node is created. Each triple encodes the information about the boolean test on the feature id f with threshold value γ in a branching node of tree T_i with the mask bitvector identifying the unreachable leaves if the associated test is false. The list of triples N_f is sorted by increasing threshold value. To make cache-friendly accesses to the data structures storing the triples (i, mask, γ) of each feature, QuickScorer stores them in three separate arrays, namely `tid[f]`, `mask[f]` and `th[f]`. The use of three distinct arrays solves some data alignment issues arising when triples of heterogeneous data types are stored contiguously in memory. For the same reason, the arrays of the different features are then juxtaposed one after the other. Finally, the leaf nodes and their score contributions are stored in a lookup table `score`. These data structures can be computed offline to represent the whole forest of regression trees and thereafter accessed in read-only mode. Figure 6.8 illustrates these data structures for the example model in Figure 6.6.



	f_1					f_2				f_3				
th	2.3	2.6	3.6	7.2	8.6	0.2	1.1	4.1	9.9	1.0	3.3	5.1	5.5	6.0
tid	1	1	1	2	2	2	1	2	1	2	2	1	2	1
mask	00000011	00111111	11110111	11001111	01111111	11011111	00001111	11111101	11011111	00111111	11110011	01111111	11110111	11111101

score									
	1	-3	-2	1	4	-7	-1	-5	T_1
	2	-2	6	-1	2	5	-9	-4	T_2

Figure 6.8: The QuickScorer data structures of the model in Figure 6.6.

The QuickScorer algorithm is detailed in Algorithm 6.1. When a document, represented by a feature vector **x** must be scored, an array of n bitvectors `exit` is created, one per tree. Every `exit` bitvector contains one bit per leaf and all bits are set to 1 (lines 1-3). Each feature is processed sequentially (lines 4-10). Whenever a false node is identified (line 6), the corresponding tree's `exit` bitvector is updated, resetting to 0 the bits of the unreachable leaves, thanks to a logical AND (∧) operation with the node's `mask` (line 7). As soon as a test evaluates to true, the remaining branching nodes cannot be false nodes, and the evaluation of the associated tests can be safely skipped. Now the score computation can take place (lines 11-15). The leftmost bit set to 1 in `exit[i]` identifies the leaf corresponding to the score contribution of tree T_i, stored in the lookup table `scores`. All score contributions are summed (weights are included in the values stored in the lookup table) and the final score `s` is returned.

Algorithm 6.1: The QuickScorer algorithm

```
   Input  : A document as an array x of $m$ input features values
   Output : The final score of the document
   QuickScorer(x):
1      exit ← an array of $n$ bitvectors of $L$ bits
2      for i ← 0 to $n-1$ do
3          exit[i] ← 11...11
4      for f ← 0 to $m-1$ do
5          k ← 0
6          while x[f] > th[f][k] do
7              exit[tid[f][k]] ← exit[tid[f][k]] ∧ mask[tid[f][k]]
8              k ← k + 1
9              if k ≥ Length(th[f]) then
10                 break
11     s ← 0
12     for i ← 0 to $n-1$ do
13         l ← the index of the leftmost bit set to 1 in exit[i]
14         s ← s + score[i,l]
16     return s
```

The original QuickScorer paper (Lucchese *et al.*, 2015b) reported up to 6.5× speedups w.r.t. VPred, depending on the number of trees and leaves. The interleaved traversal strategy of QuickScorer needs to process less nodes than in a traditional root-to-leaf visit such as in the code generation and the VPred approaches. QuickScorer exploits more efficiently the branch predictor, since the branches in Algorithm 6.1 are very easily predictable. Finally, the compact data structures and their linear access greatly improve the cache usage. In the same paper, the authors also experimented with a blocking version of QuickScorer called BWQS, adapting the strategy proposed by Tang *et al.* (2014), with a speedup of at most 1.55× w.r.t. QuickScorer on models with 20,000 trees and 64 leaves. Jin *et al.* (2016) investigated in great detail the impact of the document-based and tree-based blocking strategies on cache hierarchies, and developed an analytical cost model used to select a traversal method and blocking parameters for the effective use of memory hierarchies. In (Dato *et al.*, 2016), the QuickScorer strategy was adapted to a particular class of regression trees, called *oblivious trees*. These oblivious trees are balanced trees where, at each level, all of the branching nodes test the same feature-threshold pair (Langley and Sage, 1994; Kohavi, 1994). In (Lucchese *et al.*, 2016), a vectorised version vQS of QuickScorer is presented, exploiting the SIMD instructions of modern CPUs. The authors reported a speedup w.r.t. QuickScorer ranging from 1.2× for larger models to 3.2× for smaller models.

6.3.2 Efficient-Effective Tradeoffs in Learned Models

In the previous section, we presented the main approaches for the efficient implementation of forests of regression trees in the query processing stages with no impact on the result effectiveness. In this section, we describe some optimisation strategies for models that tradeoff efficiency improvements at the cost of some effectiveness losses.

Cambazoglu *et al.* (2010) proposed to apply the concepts of dynamic pruning discussed in Chapter 3 to query processing in additive machine learned ranking systems. They suggested to early-terminate (or *short-circuit*) the scoring process avoiding wasting time in processing documents that are unlikely to be relevant, since the relevant documents to be retrieved are typically few and only few top scoring documents are returned to the users.

In additive ensembles, document scores can be computed according to two different traversal orders, both logically structured in two nested loops. The *document-ordered traversal strategy* (DOT) loops over the documents to be scored, and for every document, it loops over the scorers, i.e., the decision trees. On the other hand, the *scorer-sorted traversal strategy* (SOT) loops over the scorers, and for every scorer, it loops over the documents. In DOT, at every iteration of the outer loop, the score of a document is completely computed, while in SOT, document scores are partially computed and accumulated at every outer loop iteration.

Cambazoglu *et al.* (2010) proposed four early exit strategies, depending on the traversal strategy. All strategies need externally-provided *thresholds*, one per scorer, and an early exit decision is taken every time a new score contribution for any document is computed.

- *early exits using score thresholds* (EST): exits are based on comparisons between accumulated scores and the provided score thresholds. After a scorer updates the accumulated score of a document, if it is less than the provided threshold, the document is not processed by any of the remaining scorers. This strategy works for both DOT and SOT. The main limitation of this strategy is that we assume that the provided score thresholds are suitable for all query-document pairs.
- *early exits using capacity thresholds* (ECT): when documents are processed under DOT,

a maximum score heap whose capacity is provided by the threshold is maintained for each scorer. Once the heap is full, a document whose partial score cannot beat the lowest partial score in the heap is discarded, otherwise the heap is updated with the new partial score and the lowest partial score stored is removed. The main limitation of this strategy is that the lowest partial score depends only on those documents already processed.

- *early exits using rank thresholds* (ERT): when documents are processed under SOT, after every scorer, the documents are ranked by their partial scores, and all documents ranked below the provided threshold are discarded. The main limitation of this strategy is that it discards documents even if their scores are just slightly smaller than the last ranked one.
- *early exits using proximity thresholds* (EPT): when documents are processed under SOT, after every scorer, the K-th document score is computed (pivot score). The following scorer processes all documents with a score higher than the pivot score, as well as those documents with smaller scores but whose score differences with the pivot score are smaller than the provided threshold. In a sense, this strategy combines both the rank and score information available thus far.

Note that the thresholds of these early exit strategies must be tuned offline, and the corresponding strategies do not guarantee the correctness of the top K results (i.e., they are unsafe, see Section 3.1). The authors reported considerable speedups when the proposed strategies were deployed using real-life documents and queries from a commercial search engine. In particular, for the EPT strategy, they reported up to 4× speedups with negligible losses in effectiveness. Another major limitation of the proposed strategies is that the costs of the scorers are assumed to be similar, while they depend on the complexity of the individual scorers and the extraction cost of the involved features.

Tuning the thresholds in the previous approaches is a first example of the effectiveness-efficiency tradeoff occurring in LTR query processing. In such cases, the tuning is performed empirically, without a well-defined and principled way to quantify such a tradeoff. Wang *et al.* (2010) proposed a novel class of *tradeoff metrics* that take into account both effectiveness and efficiency. The execution time of a query q taken from a set of queries $\mathcal{Q}$ quantifies the efficiency of its processing with the function $\sigma : \mathcal{Q} \to [0, 1]$, where 0 represents an inefficient query processing and 1 represents an efficient query processing. Effectiveness is measured with a similar function $\gamma : \mathcal{Q} \to [0, 1]$, such as one obtained from the commonly used effectiveness metrics such as MAPand NDCG. The proposed *efficiency-effectiveness tradeoff* (EET) metric for a given query q is defined as the weighted harmonic mean of $\sigma(q)$ and $\gamma(q)$:

$$\mathsf{EET}(q) = \frac{1+\beta}{\frac{1}{\sigma(q)} + \frac{\beta}{\gamma(q)}} = \frac{(1+\beta)\sigma(q)\gamma(q)}{\beta\sigma(q)+\gamma(q)}, \tag{6.2}$$

where β is a parameter controlling the relative importance of effectiveness and efficiency.[7] Given a set of queries $\mathcal{Q}$, it is possible to compute a *mean EET* metric (MEET) over the queries in the log, as follows:

$$\mathsf{MEET}(\mathcal{Q}) = \frac{1}{|Q|} \sum_{q \in \mathcal{Q}} \mathsf{EET}(q). \tag{6.3}$$

Wang *et al.* (2010) exploited the MEET measure in the LTR model learning phase. They focused on learning the weights of both the query terms and complex query operators (such as #uwλ and #1, see Section 2.2.4), in the sequential dependence proximity model. Indeed, in the weighted sequential dependence model, the weights on terms and complex operators are not constants but depend on the particular query terms and bigrams involved (Bendersky *et al.*, 2010). At query processing time, if the ratio between the bigram feature weights and the sum of the individual term feature weights is less than a threshold, the corresponding bigram feature is not used, thereby improving efficiency. The authors' experiments showed that their *feature-pruning* learned ranking functions achieve significantly decreased average query execution times with no losses in effectiveness w.r.t state-of-the-art LTR models.

Wang *et al.* (2011) proposed to implement effective yet efficient multi-stage cascading systems by progressively refining a shrinking set of candidate documents (*document-pruning*). The core idea is to process many documents at the first stages, using few and cheap features, while focusing on more computationally expensive features only at the last stages, with fewer documents. They modeled a cascade as a sequence of stages, where each stage is associated with a *pruning function* (equivalent to early exit strategies) and a *local ranking function*. Each stage receives as input the set of ranked documents from the previous stage. Each stage firstly uses the pruning function to remove documents from the input set, and then computes the score contribution of the local ranking function, updating the scores of the candidate documents still under consideration. The output results are forwarded to the next cascade stage. The goal of the cascade is to reduce the number of documents to be processed at each stage, while increasing the effectiveness of the top K documents.

The pruning functions they discussed are based on:

- *rank thresholds*: at every stage, a document is pruned if it ranks below a threshold value;
- *score thresholds*: at every stage, the document scores are linearly scaled in $[0, 1]$, and all documents below a threshold value in $[0, 1]$ are discarded;
- *mean-max thresholds*: at every stage, all documents with a score less than a linear combination of the maximum and mean scores of the document sample are pruned (the combination weights depend on the provided threshold).

The authors used a generalisation of the AdaRank (Xu and Li, 2007) boosting-based algorithm to learn the optimal sequence of ranking stages together with the pruning

[7] Originally, Wang *et al.* (2010) used β^2 instead of β as control parameter.

conditions at each stage. Differently from Cambazoglu *et al.* (2010), Wang *et al.* (2011) incorporated the selection of the pruning functions and their tuning in the machine learning algorithm. They adopted an efficiency-effectiveness tradeoff metric that can be restated using the notation of Equation (6.2) as follows:

$$\mathsf{EET}(q) = \gamma(q) + \beta\sigma(q), \tag{6.4}$$

where $\beta \in [0, 1]$ is a parameter controlling the relative importance of effectiveness and efficiency. From this tradeoff definition, as we add more stages to a cascade, the total efficiency metric $\sigma(q)$ decreases (as a query needs more time to be processed), and must be counteracted by increases in the effectiveness metric $\gamma(q)$. Wang *et al.* (2011) compared the performance of their document-pruning multi-stage cascade with their earlier feature-pruning multi-stage cascade proposed in Wang *et al.* (2010) using the weighted sequential dependence ranking function. They reported efficiency improvements up to 1.44× over the feature-tuning approach, with small gains in effectiveness.

Chen *et al.* (2017) discussed how to build efficient cost-aware cascades using gradient-boosted tree models, instead of simpler linear models. They explicitly modeled the feature extraction costs and the feature importance, and proposed a generic framework that encompasses the feature costs within the learned models. The authors noted that a cascade is composed by a sequence of increasingly complex ranking functions, where expensive features are used in later cascade stages. They explored three different feature availability settings:

- the features are sorted in ascending order of unit cost and partitioned among the stages in this order;
- the features are sorted in descending order of cost efficiency and partitioned among the stages in this order;[8]
- all features are available among all stages.

Their approach was compared w.r.t. the document-pruning approach of (Wang *et al.*, 2011), using both linear and tree-based models with different feature allocation strategies. The experiments showed that the proposed approach can consistently achieve better tradeoffs than the document-pruning approach, even though their approach to parameter selection for the learned models is largely empirical and costly.

6.4 Summary

This chapter introduced the separation of the ranking process into cascades, dictated by the necessity of both effectively and efficiently combining large numbers of relevance signals with learning-to-rank algorithms and models. We presented and discussed the three main

[8] The cost efficiency of a feature is defined as its importance score divided by its unit cost.

factors impacting the efficiency of LTR model applications, namely: the initial base ranking stage, the calculation/extraction of the features, and the efficient application of the learned models to combine those features into a final improved ranking.

It remains to be seen how the advent of deep neural IR techniques will change the learning-to-rank paradigm. Thus far, as mentioned in Section 6.2, many deep learning approaches can be applied as query dependent features within a learning-to-rank framework, and we have not, thus far, witnessed changes to the learning-to-rank paradigm due to deep learning.

7 Open Directions & Conclusions

In this chapter, we provide concluding remarks on the conducted survey and discuss future open directions for the efficient deployment of the query processing component in information retrieval systems.

7.1 Summary

This monograph aimed to both provide the foundations of query processing, as well as to discuss more recent trends. In particular, in Chapter 2, we provided the necessary background material such as data structures, posting lists compression and skipping, while also introducing the core concepts of TAAT and DAAT. We believe that they were necessary to cover in order to provide all readers with a coherent grounding in these concepts.

Chapter 3 introduced dynamic pruning techniques, including MaxScore, WAND and BMW. In particular, the latter forms the state-of-the-art for dynamic pruning, and can result in an 8× improvement over a basic DAAT implementation. These algorithms are complex in nature, have been described in different papers, and have been sometimes interpreted or presented in different ways. We strongly believe that our descriptions of all of these techniques, using the same notations and abstractions, will allow readers to quickly grasp their key attributes, their intricacies and differences, as well as providing a fast basis for their implementations.

Chapter 4 described a comparatively modern development in the form of query efficiency predictors (QEPs), and their applications. All of the described applications are intended to make on-the-fly adaptations to the search engine's processing of a query, based on how long the query is expected to take, through the use of QEP s, to reduce the overall response times. We described 5 QEP applications from the literature including the selective adjustment of the pruning aggressiveness and the selective parallelisation of long-running queries.

Chapter 5 described impact-ordered posting lists, and the resulting SAAT dynamic pruning techniques designed to work with such index layouts. They have advantages over docid-ordered posting lists, in that stopping the retrieval of a given query early is less likely to impact upon the resulting effectiveness.

Chapter 6 discussed the efficient application and deployment of learning-to-rank models in a search setting. This encompassed the necessary infrastructure to compute the features on a retrieved candidate set of documents, as well as the efficient application of complex tree-based learned models to generate the final scores for the candidate documents. We also highlighted recent trends in varying the configuration of dynamic pruning techniques used to compute the candidate set, for example by reducing the number of documents for long-running queries.

This Chapterprovides additional highlights towards open directions and emerging trends in efficient information retrieval, namely: the use of signatures (section 7.2) as probabilistic approaches for retrieving documents matching a query; techniques that directly target a reduction in the power consumption of search engines (section 7.3); new efficiency approaches made possible by new hardware architectures (section 7.4) or implemented on new software paradigms (section 7.5); and, finally, search architectures targeting real-time search settings, where results need to be constantly up-to-date (section 7.6).

7.2 Signatures

Almost all of the approaches described in this monograph use the classical inverted index data structure to support retrieval. An alternative that was examined and discarded in the classical literature (Zobel *et al.*, 1998) is the use of signature files. In signatures, each term of a document is hashed a number of times, to determine the bits of a document signature that should be set. Queries are similarly hashed, and matching occurs by comparing the query signature to the signature of each document. Some hash collisions will occur, meaning that each possible matching document must be checked against the query to determine whether it is a false match (terms do not occur in the document), or a true match.

Of course, storing and searching signatures for each document is both space- and time-inefficient. In particular, every document's signature must be scanned, even if a term is very rare. Bit-slicing allows multiple documents to be searched simultaneously, as well as simplifying the necessary bitwise operations. Bit-sliced *block* signatures work by assigning multiple documents to each bit in a signature, however, Zobel *et al.* (1998) did not find that this approach offered much benefits. In contrast, and more recently, Goodwin *et al.* (2017) described BitFunnel, a signature-based search engine implementation based on Bloom filters, and proposed new signature-based layouts: *frequency-conscious signatures* to reduce the memory footprint, and *higher-rank* rows of signatures. In particular, frequency-conscious signatures vary the number of hash functions on a term-by-term basis within the same Bloom filter. Higher-rank rows generalises the ideas behind blocking such that each term simultaneously hashes to multiple bit-sliced signatures with different blocking factors. A Bloom filter on n bits for a given term is the rank-0 row. It is “folded”, pairing the low bits with the high bits, into a $\lceil n/2 \rceil$ bits rank-1 row, and so on. Given a fixed bit density (i.e., number of bit sets w.r.t. the total number of bits), each term is associated with the higher-rank row with the closest bit density, and rows with a rank higher than 0 are “unfolded” at runtime for query processing.

Experiments conducted in comparison to a Partitioned Elias-Fano (PEF, described on page 19) index, showed that, depending on the density of the shard of the index, query throughput could be improved from 1.5× for smaller documents upto 8× for longer documents using the BitFunnel signature files, at the expense of upto $2-5$ times more index space, and false positive rates of $1-4\%$.

BitFunnel has been deployed at Bing since 2013 across thousands of servers, and improved server query capacity by a factor of 10 (Goodwin *et al.*, 2017). This has spawned further research in how signatures can be applied in Web search (e.g., (Liu *et al.*, 2018)), and we expect this trend to continue.

7.3 Energy Efficiency

The infrastructure of a Web search engine can be considered at different levels, from the single search server (sometimes called an index serving node, or ISN), to a cluster of ISNs responsible for storing the data and processing queries, up to the level of data centres, which can be geographically distributed around the world. The techniques that we have discussed in this survey focus on those at the level of the individual ISN, but which can be deployed across many servers to emphasise their particular advantages. In general, efficiency savings at the level of the individual server without (significant) effectiveness loss allow a potential reduction in the number of ISNs required to run the search engine. For instance, Jeon *et al.* (2013) claimed that their QEP-based selective parallelisation strategy (described in Section 4.3.3) could reduce the number of query servers necessary to run Microsoft Bing by one third.

Yet, the distributed nature of search engine infrastructures brings further possibilities to reduce energy consumption that have not been well investigated in the literature. The impact of new solutions on this front could have a marked impact on the economic profitability of a Web search engine, and, at the same time, could decrease its pollution impact on the environment. Moreover, the carbon footprint of IT infrastructures is likely to continue to grow in importance in future years, as strategic decisions at government and international levels continue to impose further constraints and expectations on the sustainability and the eco-friendliness of IT systems.

In the past, a large part of the energy consumption of a data centre was accounted for by inefficiencies in its cooling and power supply systems. However, careful design of the data centre can drastically reduce the energy wastage of those infrastructures. In fact, now, the CPUs of the servers in the data centres are the main energy consumers (Barroso *et al.*, 2013).

The energy efficiency of a Web search engine can be improved at the levels of: (i) the search server, (ii) the search cluster, and (iii) across multiple data centres. At the search server level, most approaches to energy efficiency solutions rely on *energy-proportional computing*, i.e., hardware components with power consumption proportional to utilisation (Barroso and Hölzle, 2007). For instance, Modern CPUs expose multiple frequencies available to the CPU cores. Indeed, a CPU core can operate at different clock frequencies (e.g., 800 MHz, 1.6 GHz, 2.1 GHz, etc.). This is possible thanks to Dynamic Frequency and Voltage Scaling (DVFS) technologies (Snowdon *et al.*, 2005). Higher frequencies correspond to a higher performance and consumption, while lower frequencies correspond to a lower performance and consumption. Catena *et al.* (2015) proposed to perform frequency throttling according to the query server utilisation, i.e., the ratio between the query arrival rate and the query processing rate. This conservative approach was later refined by Catena and Tonellotto (2017), leveraging the fact that users can hardly notice response times that are faster than their expectations (Arapakis *et al.*, 2014). They proposed the Predictive Energy Saving Online Scheduling algorithm (PESOS, described on page 73), which considers the latency requirement of queries as an explicit parameter, and tries to process queries no faster than required. In doing so, the CPU's energy consumption is reduced while respecting each query's latency constraint. Their experiments showed that while the conservative approach can reduce the energy consumption of a CPU core by more than 40%, but with uncontrollable latency violations, PESOS can reduce energy consumption by 20% up to 30%, while respecting the required tail latency.

At the level of the search cluster, both workload consolidation and power capping can lead to energy savings. Freire *et al.* (2014) and Freire *et al.* (2015) proposed a self-adaptive model to manage the number of active search servers in a replicated search engine, while guaranteeing acceptable response times. By exploiting the historical and current query loads, their model autonomously decides whether to activate a search server or put it on standby. The latter option allows to reduce the energy consumption of the system during low query loads, while the former allows to increase the system performance when the system faces a high query volume. Simulation results showed that the proposed model reduces by 33% the search engine energy consumption, with respect to a naive baseline where all search servers are always active. At the same time, the authors observed only little increases in query response times and small percentages of unanswered queries, i.e., queries that are not processed within an acceptable time since their arrival. Lo *et al.* (2014) introduced PEGASUS, a feedback-based model that dynamically caps the CPUs power consumption of a distributed search engine. Essentially, PEGASUS constantly monitors the search engine latency and passes this value to a centralised rule engine. Depending on the observed latency, the rule engine decides whether to increase or decrease the CPUs performance by exploiting DVFS technologies. Experimenting on a Google production cluster, the authors observed a 20% power consumption reduction and estimated that a distributed version of PEGASUS could nearly double those savings. Catena *et al.* (2018) evaluated both PESOS (see Section 4.3.5) and PEGASUS on a simulated distributed Web search engine composed of a thousand of servers. Their results showed that PESOS can reduce the CPU energy consumption of a distributed Web search engine by up to 18% with respect to PEGASUS, while providing query response times that are in-line with user expectations.

To improve the energy efficiency of multi-center search engines, it has also been proposed to leverage spatial and temporal variations in both the energy prices and query workloads. Due to time zone differences, a search engine's data center may experience a high workload and energy price at a given moment, while other distant sites are under-utilised and can use cheaper electricity. Thus, the first data center could forward its queries to other sites to reduce its energy expenditure. However, network latencies have to be carefully considered, to ensure that acceptable query response times are not exceeded. Moreover, data centers have limited processing capacity. It is therefore not possible to forward too many queries towards a particular site. Kayaaslan *et al.* (2011) investigated the possibility to dynamically shift the query workload among data centers by using *query forwarding*. Simulation results showed that multi-center search engines can save up to 35% in energy expenditure, when compared to a system that always locally solves its incoming queries. Similarly, Teymorian *et al.* (2013) proposed the Rank-Energy Selective Query forwarding (RESQ) algorithm, which decides when and where to forward a query by modeling the problem as a linear program, obtaining similar results in term of energy efficiency. Blanco *et al.* (2016) investigated the potential benefits of query forwarding when renewable sources of green energy are available at different data centres. They designed an algorithm, which decides what fraction of the incoming query load arriving into one processing facility must be forwarded to be processed at different sites to optimise the use of available green energy sources. Their experiments with a real query traffic from a large search engine showed that the proposed solution maintains a high query throughput, while reducing by up to ∼25% the energy operational costs of multi-center search engines.

The energy efficiency of data centers and, in particular, of the Web search engines they host, is becoming more and more important. Their economic impact, as well as their global emission footprint, is, and will continue to be, of a growing concern (Belkhir and Elmeligi (2018) stated that data centers will account for 45% of the ICT global carbon footprint by 2020), and therefore we expect that research in this area will continue to intensify in the next few years.

7.4 Modern Hardware Architectures

With the scale of the search engines' data centres, it is only natural that customised hardware solutions are increasingly being developed to suit the particular workload of a search engine. For instance, in 2012, Google was developing customised networking switches[1] for their data centres. Since then, the added-value and suitability of new hardware architectures, such as SSDs, GPUs and FPGA are the object of new research trends.[2]

In particular, solid state drives (SSDs) offer larger storage mediums that support better random access than hard disk drives. Wang *et al.* (2013) discussed how this can be used to benefit search engine caching strategies; Risvik *et al.* (2013) described the use of SSDs for storing posting lists for phrases and term pair co-occurrences – while the posting list of a phrase may be much shorter than the constituent terms, there can be many more phrases to store. Using SSDs allow more posting lists to be stored than can be fitted in RAM.

Reconfigurable chips, such as Field Programmable Gate Arrays (FPGAs), are considered to be viable accelerators for various core functions in Web search engines (Putnam *et al.*, 2014). FGPAs developed as part of Microsoft's Project Catapult [3] are deployed in the Bing data centers (Culpepper *et al.*, 2018). Each FPGA is designed to be resilient and to boost the query throughput, and is responsible for feature extraction and the execution of learning-to-rank algorithms, receiving as input first-stage results from a set of query processing nodes. These FPGAs have increased the ranking throughput in a production search infrastructure by up to 95% at comparable latencies to a software-only solution (Putnam *et al.*, 2014). We expect that distributed reconfigurable infrastructures incorporating FPGAs will remain crucial in the future for continued cost and capability improvements.

GPUs have clearly revolutionised the availability of deep neural network learning at both low time and low costs. They also have applicability to tasks such as posting list intersection (Wu *et al.*, 2010), and learning-to-rank model traversal (Lettich *et al.*, 2018), to name but a few. Google has a customised chip specifically designed for deep neural network learning denoted as a Tensor Processing Unit (TPU),[4] which has benefited many of its products including Search.[5]

Overall, as hardware engineers carry on developing new specialised architectures, these are likely to continue to benefit search applications, which are notably among some of the world's most largest users of IT infrastructures. Moreover, these hardware architectures will soon be available on all users' clients (from phone to tablet to laptop to desktop), providing opportunities to off-load work from the data center and onto the user's device (Culpepper

[1] https://www.wired.com/2012/09/pluto-switch/

[2] https://www.microsoft.com/en-us/research/project/project-catapult/

[3] https://www.microsoft.com/en-us/research/project/project-catapult/

[4] https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[5] https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html

et al., 2018).

7.5 Clouds and New Software Paradigms

Cloud-based data centres and software architectures have changed the nature of many software deployments, and are now beginning to impact upon the IR system architectures. For instance, cloud services have evolved from heavy-weight virtual machine instances to lighter-weight containers (e.g. Docker), towards lightweight "serverless" microservices deployed as Lambda functions (e.g. AWS Lambdas). Such lambda functions are stateless, but allow a cost benefit due to a charging model that only requires payment for each invocation. In fact, such lambda functions have been shown to be usable for large-scale data analytics, e.g.Spark (Kim and Lin, 2018). We note that the initial experiments by Crane and Lin (2017) towards deploying a search service built upon AWS Lambdas, and subsequent attempts at neural ranking models entirely built on such serverless infrastructures (Tu *et al.*, 2018). There have also been attempts at implementing search engines that run in the ever more powerful browser-based Javascript engines Lin (2015).

Overall, since the invention & implementation of MapReduce by Google in the early 2000s to make a scalable search indexing pipeline, it is clear that new software paradigms are closely tied to search infrastructures, and we expect innovations to continue to drive how we think about and implement search in the future. Moreover, the increasing evolution of search engines from a first-order technology (directly accessed by users) to a component in a larger microservice ecosystem – for instance as part of the infrastructure underlying a conversational search agent – will continue to drive new research into suitable architectures and data-structures to enable effective, efficient and frictionless pipeline integration (Kyriazis *et al.*, 2018).

7.6 Real-Time and Social Media Search

This survey has focussed on search engine deployments that are mostly static in nature, i.e., where the corpus is not frequently changing. In fact, specialisations of the standard index data structures are needed to handle cases where the index is being maintained with continuous document additions, updates and deletions. For instance, the compressed nature of the posting lists are not naturally amenable to the additions.

Index maintenance of such events is typically handled by the use of uncompressed index shards, which have posting lists that can easily be appended to. Such index shards can be held in-memory, and then written to disk once they are full. Index shards should be merged, to ensure efficient retrieval. Strohman and Croft (2006) described Indri's implementation, while arguing that the merging should occur in exponentially growing shard sizes.

The rise of social media has focused particular research attention into the challenges of searching voluminous social media streams. For instance, when searching Twitter, it is likely that the user would want some of the most recently posted tweets that match their query. This necessitates a low-latency indexing of tweets, whereby these tweets must be available for searching almost as soon as they are posted. Indeed, it is most likely that the newest tweets should be retrieved first since they are most likely to be of high value.

Busch *et al.* (2012) described the index layout of Twitter's Lucene-based search infrastructure called Earlybird in 2011-2012. Earlybird uses several important observations in its design: firstly, due to the short length of tweet documents, within-document term frequencies are rarely greater than 1, and hence a posting only needs to contain the docid and the term position; each occurrence obtains another posting entry. These are stored in fixed length representations in contiguous memory arrays: 24-bits are devoted to storing the document docid, and 8 bits for the term position (a tweet cannot have more than $2^8 = 256$ words). Moreover, since the index is not compressed, there is no need for skip-lists because the array can be directly binary-searched to identify the posting(s) for a given document.

At any point in time there is an *active* index shard in Earlybird that is responsible for indexing new tweets. The space allocation for the posting lists of terms is notable, in that terms are progressively allocated exponentially larger blocks of memory. Once the active index shard is full (approx $2^{24} = 16M$ tweets), it becomes *read only* and is further compressed and optimised for fast reading. The memory allocation for posting lists was further investigated by (Asadi *et al.*, 2013), while the same authors also investigated, for tweet search, the use of Bloom filters for candidate generation before applying a learned model.

Finally, we note that social media is not the only form of streaming data that is necessary to search in real-time. Increasingly, the world is becoming more connected, with the Internet-of-Things (IoT) connected devices. The plethora of devices producing data leads to the emergence of new information seeking tasks that necessarily require new types of search infrastructures (Culpepper *et al.*, 2018; McCreadie *et al.*, 2016). For instance, venue recommendation describes the task of making personalised suggestions of points-of-interests for a user to visit. This can benefit from up-to-date information about the busy-ness of venues (Deveaud *et al.*, 2015), as might be obtained from social sensors such as Foursquare, or from physical sensing IoT networks (such as people's detectors based on WiFi or mobile phone cell usage). New information-seeking tasks arising from emerging IoT networks will continue to necessitate new types of search infrastructures.

Acknowledgements

We would like to thank Maarten de Rijke for his patience and encouragements during the preparation of this manuscript, as well as the three anonymous reviewers for their constructive suggestions and thoughtful comments towards improving the manuscript.

Nicola Tonellotto acknowledges the partial support by the BIGDATAGRAPES (grant agreement No. 780751) project, which has received funding from the European Union's Horizon 2020 research and innovation framework, within the Information and Communication Technologies work programme.

References

Amati, G. and C. J. Van Rijsbergen (2002). “Probabilistic Models of Information Retrieval Based on Measuring the Divergence from Randomness”. *ACM Trans. Inf. Syst.* 20(4): 357–389. ISSN: 1046-8188. DOI: 10.1145/582415.582416.

Anh, V. N., O. de Kretser, and A. Moffat (2001). “Vector-space Ranking with Effective Early Termination”. In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 35–42. ISBN: 1-58113-331-6. DOI: 10.1145/383952.383957.

Anh, V. N. and A. Moffat (1998). “Compressed Inverted Files with Reduced Decoding Overheads”. In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 290–297. ISBN: 1-58113-015-5. DOI: 10.1145/290941.291011.

Anh, V. N. and A. Moffat (2002). “Impact Transformation: Effective and Efficient Web Retrieval”. In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 3–10. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564380.

Anh, V. N. and A. Moffat (2005a). “Inverted Index Compression Using Word-Aligned Binary Codes”. *Inf. Retr.* 8(1): 151–166. ISSN: 1386-4564. DOI: 10.1023/B:INRT.0000048490.99518.5c.

Anh, V. N. and A. Moffat (2005b). “Simplified Similarity Scoring Using Term Ranks”. In: *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 226–233. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076075.

Anh, V. N. and A. Moffat (2006a). “Pruned Query Evaluation Using Pre-computed Impacts”. In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 372–379. ISBN: 1-59593-369-7. DOI: 10.1145/1148170.1148235.

Anh, V. N. and A. Moffat (2006b). “Pruning Strategies for Mixed-mode Querying”. In: *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*. ACM. 190–197. ISBN: 1-59593-433-2. DOI: 10.1145/1183614.1183645.

Anh, V. N. and A. Moffat (2006c). “Structured Index Organizations for High-Throughput Text Querying”. In: *Proceedings of the 13th International Conference on String Processing and Information Retrieval*. Springer. 304–315. ISBN: 978-3-540-45775-6. DOI: 10.1007/11880561_25.

Arapakis, I., X. Bai, and B. B. Cambazoglu (2014). “Impact of Response Latency on User Behavior in Web Search”. In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 103–112. ISBN: 978-1-4503-2257-7. DOI: 10.1145/2600428.2609627.

Arroyuelo, D., S. González, M. Marin, M. Oyarzún, and T. Suel (2012). "To Index or Not to Index: Time-space Trade-offs in Search Engines with Positional Ranking Functions". In: *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 255–264. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348320.

Asadi, N. and J. Lin (2012). "Fast Candidate Generation for Two-phase Document Ranking: Postings List Intersection with Bloom Filters". In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. ACM. 2419–2422. ISBN: 978-1-4503-1156-4. DOI: 10.1145/2396761.2398656.

Asadi, N. and J. Lin (2013). "Document Vector Representations for Feature Extraction in Multi-stage Document Ranking". *Inf. Retr.* 16(6): 747–768. ISSN: 1386-4564. DOI: 10.1007/s10791-012-9217-9.

Asadi, N., J. Lin, and M. Busch (2013). "Dynamic Memory Allocation Policies for Postings in Real-time Twitter Search". In: *Proceedings of the 19th ACM International Conference on Knowledge Discovery and Data Mining*. ACM. 1186–1194. ISBN: 978-1-4503-2174-7. DOI: 10.1145/2487575.2488221.

Asadi, N., J. Lin, and A. P. de Vries (2014). "Runtime Optimizations for Tree-Based Machine Learning Models". *IEEE Trans. Knowl. Data Eng.* 26(9): 2281–2292.

Baeza-Yates, R. and B. Ribeiro-Neto (2008). *Modern Information Retrieval (2nd ed.)* Addison-Wesley. ISBN: 9780321416919.

Barbay, J., A. López-Ortiz, and T. Lu (2006). "Faster Adaptive Set Intersections for Text Searching". In: *Experimental Algorithms*. Springer. 146–157. ISBN: 978-3-540-34598-5. DOI: 10.1007/11764298_13.

Barroso, L. A., J. Clidaras, and U. Hölzle (2013). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2nd ed.)* Morgan & Claypool Publishers. ISBN: 9781627050098.

Barroso, L. A. and U. Hölzle (2007). "The Case for Energy-Proportional Computing". *Computer*. 40(12): 33–37. ISSN: 0018-9162. DOI: 10.1109/MC.2007.443.

Bartell, B. T., G. W. Cottrell, and R. K. Belew (1994). "Automatic Combination of Multiple Ranked Retrieval Systems". In: *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Springer. 173–181. ISBN: 0-387-19889-X.

Belkhir, L. and A. Elmeligi (2018). "Assessing ICT Global Emissions Footprint: Trends to 2040 & recommendations". *Journal of Cleaner Production*. 177: 448–463. ISSN: 0959-6526. DOI: 10.1016/j.jclepro.2017.12.239.

Bendersky, M., W. B. Croft, and Y. Diao (2011). "Quality-biased Ranking of Web Documents". In: *Proceedings of the 4th ACM International Cconference on Web Search and Data Mining*. ACM. 95–104. ISBN: 978-1-4503-0493-1. DOI: 10.1145/1935826.1935849.

Bendersky, M., D. Metzler, and W. B. Croft (2010). "Learning Concept Importance Using a Weighted Dependence Model". In: *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*. ACM. 31–40. ISBN: 978-1-60558-889-6. DOI: 10.1145/1718487.1718492.

Blanco, R., M. Catena, and N. Tonellotto (2016). "Exploiting Green Energy to Reduce the Operational Costs of Multi-Center Web Search Engines". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 1237–1247. ISBN: 978-1-4503-4143-1. DOI: 10.1145/2872427.2883021.

Boldi, P. and S. Vigna (2005). "Compressed Perfect Embedded Skip Lists for Quick Inverted-index Lookups". In: *Proceedings of the 12th International Conference on String Processing and Information Retrieval*. Springer. 25–28. ISBN: 978-3-540-32241-2. DOI: 10.1007/11575832_3.

Bonacic, C., C. Garcia, M. Marin, M. Prieto, F. Tirado, and C. Vicente (2008). "Improving Search Engines Performance on Multithreading Processors". In: *High Performance Computing for Computational Science - VECPAR 2008*. Ed. by J. M. L. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. C. Lopes. Springer. 201–213. ISBN: 978-3-540-92859-1.

Bookstein, A., S. T. Klein, and T. Raita (1997). "Modeling Word Occurrences for the Compression of Concordances". *ACM Trans. Inf. Syst.* 15(3): 254–290. ISSN: 1046-8188. DOI: 10.1145/256163.256166.

Bortnikov, E., D. Carmel, and G. Golan-Gueta (2017). "Top-k Query Processing with Conditional Skips". In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 653–661. ISBN: 978-1-4503-4914-7. DOI: 10.1145/3041021.3054191.

Bosch, A. van den, T. Bogers, and M. de Kunder (2016). "Estimating Search Engine Index Size Variability: a 9-year Longitudinal Study". *Scientometrics*. 107(2): 839–856. ISSN: 1588-2861. DOI: 10.1007/s11192-016-1863-z.

Breiman, L. (2001). "Random Forests". *Machine Learning*. 45(1): 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324.

Broccolo, D., C. Macdonald, S. Orlando, I. Ounis, R. Perego, F. Silvestri, and N. Tonellotto (2013). "Load-sensitive Selective Pruning for Distributed Search". In: *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. ACM. 379–388. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505699.

Broder, A. Z., D. Carmel, M. Herscovici, A. Soffer, and J. Zien (2003). "Efficient Query Evaluation using a Two-level Retrieval Process". In: *Proceedings of the 12th International Conference on Information and Knowledge Management*. ACM. 426–434. ISBN: 1-58113-723-0. DOI: 10.1145/956863.956944.

Brown, E. W. (1995). “Fast Evaluation of Structured Queries for Information Retrieval”. In: *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 30–38. ISBN: 0-89791-714-6. DOI: 10.1145/215206.215329.

Brutlag, J. and E. Schuman (2009). *Performance Related Changes and their User Impact*.

Buckley, C. and A. F. Lewit (1985). “Optimization of Inverted Vector Searches”. In: *Proceedings of the 8 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 97–110. ISBN: 0-89791-159-8. DOI: 10.1145/253495.253515.

Burges, C., T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender (2005). “Learning to Rank Using Gradient Descent”. In: *Proceedings of the 22nd International Conference on Machine Learning*. ACM. 89–96. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102363.

Busch, M., K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin (2012). “Earlybird: Real-Time Search at Twitter”. In: *Proceedings of the 28th IEEE International Conference on Data Engineering*. IEEE. 1360–1369. ISBN: 978-0-7695-4747-3. DOI: 10.1109/ICDE.2012.149.

Büttcher, S., C. Clarke, and G. V. Cormack (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press. ISBN: 0262026511.

Cambazoglu, B. B. and C. Aykanat (2006). “Performance of Query Processing Implementations in Ranking-based Text Retrieval Systems Using Inverted Indices”. *Inf. Process. Manage.* 42(4): 875–898. ISSN: 0306-4573. DOI: 10.1016/j.ipm.2005.06.004.

Cambazoglu, B. B., H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt (2010). “Early Exit Optimizations for Additive Machine Learned Ranking Systems”. In: *Proceedings of the 3rd ACM international conference on Web search and data mining*. ACM. 411–420. ISBN: 978-1-60558-889-6. DOI: 10.1145/1718487.1718538.

Cambazoglu, B. B. and R. A. Baeza-Yates (2015). *Scalability Challenges in Web Search Engines*. Morgan & Claypool Publishers.

Cao, Z., T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li (2007). “Learning to Rank: From Pairwise Approach to Listwise Approach”. In: *Proceedings of the 24th International Conference on Machine Learning*. ACM. 129–136. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273513.

Capannini, G., C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, and N. Tonellotto (2016). “Quality versus Efficiency in Document Scoring with Learning-to-Rank Models”. *Inf. Process. Manage.* 52(6): 1161–1177. ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2016.05.004.

Carmel, D., D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer (2001). “Static Index Pruning for Information Retrieval Systems”. In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 43–50. ISBN: 1-58113-331-6. DOI: 10.1145/383952.383958.

Carmel, D. and E. Yom-Tov (2010). *Estimating the Query Difficulty for Information Retrieval*. Morgan & Claypool Publishers. ISBN: 978-1-4503-0153- 4.

Catena, M. and N. Tonellotto (2017). “Energy-Efficient Query Processing in Web Search Engines”. *IEEE Trans. Knowl. Data Eng.* 29(7): 1412–1425. ISSN: 1041-4347. DOI: 10.1109/TKDE.2017.2681279.

Catena, M., O. Frieder, and N. Tonellotto (2018). “Efficient Energy Management in Distributed Web Search”. In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM. 1555–1558. ISBN: 978-1-4503-6014-2. DOI: 10.1145/3269206.3269263.

Catena, M., C. Macdonald, and N. Tonellotto (2015). “Load-sensitive CPU Power Management for Web Search Engines”. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 751–754. ISBN: 978-1-4503-3621-5. DOI: 10.1145/2766462.2767809.

Chakrabarti, K., S. Chaudhuri, and V. Ganti (2011). “Interval-based Pruning for Top-k Processing over Compressed Lists”. In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. IEEE. 709–720. ISBN: 978-1-4244-8959-6. DOI: 10.1109/ICDE.2011.5767855.

Chapelle, O. and Y. Chang (2011). “Yahoo! Learning to Rank Challenge Overview”. *Journal of Machine Learning Research-Proceedings Track*. 14: 1–24.

Chapelle, O., D. Metlzer, Y. Zhang, and P. Grinspan (2009). “Expected Reciprocal Rank for Graded Relevance”. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM. 621–630. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646033.

Chen, R.-C., L. Gallagher, R. Blanco, and J. S. Culpepper (2017). “Efficient Cost-Aware Cascade Ranking in Multi-Stage Retrieval”. In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 445–454. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3080819.

Chierichetti, F., S. Lattanzi, F. Mari, and A. Panconesi (2008). “On Placing Skips Optimally in Expectation”. In: *Proceedings of the 1st International Conference on Web Search and Data Mining*. ACM. 15–24. ISBN: 978-1-59593-927-2. DOI: 10.1145/1341531.1341537.

Clarke, C. L., J. S. Culpepper, and A. Moffat (2016). “Assessing Efficiency-Effectiveness Tradeoffs in Multi-stage Retrieval Systems without using Relevance Judgments”. *Inf. Retr.* 19(4): 351–377. ISSN: 1386-4564. DOI: 10.1007/s10791-016-9279-1.

Crane, M., J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman (2017). “A Comparison of Document-at-a-Time and Score-at-a-Time Query Evaluation”. In: *Proceedings of the 10th ACM International Conference on Web Search and Data Mining*. ACM. 201–210. ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018726.

Crane, M. and J. Lin (2017). “An Exploration of Serverless Architectures for Information Retrieval”. In: *Proceedings of the International Conference on Theory of Information Retrieval*. ACM. 241–244. ISBN: 978-1-4503-4490-6. DOI: 10.1145/3121050.3121086.

Crane, M., A. Trotman, and R. O’Keefe (2013). “Maintaining Discriminatory Power in Quantized Indexes”. In: *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*. ACM. 1221–1224. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2507860.

Croft, W. B., D. Metzler, and T. Strohman (2009). *Search Engines: Information Retrieval in Practice*. Addison-Wesley. ISBN: 0136072240.

Cronen-Townsend, S., Y. Zhou, and W. B. Croft (2002). “Predicting Query Performance”. In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 299–306. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564429.

Culpepper, J. S., C. L. A. Clarke, and J. Lin (2016). “Dynamic Cutoff Prediction in Multi-Stage Retrieval Systems”. In: *Proceedings of the 21st Australasian Document Computing Symposium*. ACM. 17–24. ISBN: 978-1-4503-4865-2. DOI: 10.1145/3015022.3015026.

Culpepper, J. S., F. Diaz, and M. D. Smucker (2018). “Research Frontiers in Information Retrieval: Report from the Third Strategic Workshop on Information Retrieval in Lorne (SWIRL 2018)”. *SIGIR Forum*. 52(1): 34–90. ISSN: 0163-5840. DOI: 10.1145/3274784.3274788.

Culpepper, J. S. and A. Moffat (2010). “Efficient Set Intersection for Inverted Indexing”. *ACM Trans. Inf. Syst.* 29(1): 1:1–1:25. ISSN: 1046-8188. DOI: 10.1145/1877766.1877767.

Dang, V., M. Bendersky, and W. B. Croft (2013). “Two-Stage Learning to Rank for Information Retrieval”. In: *Proceedings of the 35th European Conference on IR Research*. Springer. 423–434. ISBN: 978-3-642-36972-8. DOI: 10.1007/978-3-642-36973-5_36.

Daoud, C. M., E. S. de Moura, A. Carvalho, A. S. da Silva, D. Fernandes, and C. Rossi (2016). “Fast Top-k Preserving Query Processing using Two-tier Indexes”. *Inf. Process. Manage.* 52(5): 855–872. ISSN: 0306-4573. DOI: http://dx.doi.org/10.1016/j.ipm.2016.03.005.

Dato, D., C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini (2016). “Fast Ranking with Additive Ensembles of Oblivious and Non-Oblivious Regression Trees”. *ACM Trans. Inf. Syst.* 35(2): 15:1–15:31. ISSN: 1046-8188. DOI: 10.1145/2987380.

Dean, J. (2009). “Challenges in Building Large-scale Information Retrieval Systems: Invited Talk”. In: *Proceedings of the 2nd ACM International Conference on Web Search and Data Mining*. ACM. 1–1. ISBN: 978-1-60558-390-7. DOI: 10.1145/1498759.1498761.

Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman (1990). "Indexing by Latent Semantic Analysis". *J. Am. Soc. Inf. Sc.* 41(6): 391–407. ISSN: 1097-4571. DOI: 10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

Dehghani, M., H. Zamani, A. Severyn, J. Kamps, and W. B. Croft (2017). "Neural Ranking Models with Weak Supervision". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 65–74. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3080832.

Demaine, E. D., A. López-Ortiz, and J. I. Munro (2000). "Adaptive Set Intersections, Unions, and Differences". In: *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 743–752. ISBN: 0-89871-453-2.

Deveaud, R., M.-D. Albakour, C. Macdonald, and I. Ounis (2015). "Experiments with a Venue-Centric Model for Personalisedand Time-Aware Venue Suggestion". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM. 53–62. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806484.

Dimopoulos, C., S. Nepomnyachiy, and T. Suel (2013a). "A Candidate Filtering Mechanism for Fast Top-k Query Processing on Modern Cpus". In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 723–732. ISBN: 978-1-4503-2034-4. DOI: 10.1145/2484028.2484087.

Dimopoulos, C., S. Nepomnyachiy, and T. Suel (2013b). "Optimizing Top-k Document Retrieval Strategies for Block-max Indexes". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. ACM. 113–122. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433412.

Ding, S. and T. Suel (2011). "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*. ACM. 993–1002. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2010048.

Diriye, A., R. White, G. Buscher, and S. Dumais (2012). "Leaving So Soon?: Understanding and Predicting Web Search Abandonment Rationales". In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. ACM. 1025–1034. ISBN: 978-1-4503-1156-4. DOI: 10.1145/2396761.2398399.

Doszkocs, T. E. (1982). "From Research to Application: The Cite Natural Language Information Retrieval System". In: *Proceedings of the 5th Annual ACM Conference on Research and Development in Information Retrieval*. Springer. 251–262. ISBN: 0-387-11978-7.

Elias, P. (1974). "Efficient Storage and Retrieval by Content and Address of Static Files". *J. ACM*. 21(2): 246–260. ISSN: 0004-5411. DOI: 10.1145/321812.321820.

Elias, P. (1975). "Universal Codeword Sets and Representations of the Integers". *IEEE Trans. Inf. Theory*. 21(2): 194–203. ISSN: 0018-9448. DOI: 10.1109/TIT.1975.1055349.

Fano, R. M. (1971). "On the Number of Bits Required to Implement an Associative Memory". *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA*.

Fontoura, M., V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien (2011). "Evaluation Strategies for Top-k Queries over Memory-Resident Inverted Indexes". *Proc. VLDB Endow.* 4(12): 1213–1224.

Fox, E. A. and J. A. Shaw (1994). "Combination of Multiple Searches". In: *Proceedings of the 2nd Text REtrieval Conference (TREC-2)*. NIST. 243–252.

Frachtenberg, E. (2009). "Reducing Query Latencies in Web Search Using Fine-Grained Parallelism". *World Wide Web*. 12(4): 441. ISSN: 1573-1413. DOI: 10.1007/s11280-009-0066-4.

Freire, A., C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda (2012). "Scheduling Queries Across Replicas". In: *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 1139–1140. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348508.

Freire, A., C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda (2013). "Hybrid Query Scheduling for a Replicated Search Engine". In: *Proceedings of the 35th European Conference on IR Research*. Springer. 435–446. ISBN: 978-3-642-36973-5. DOI: 10.1007/978-3-642-36973-5_37.

Freire, A., C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda (2014). "A Self-adapting Latency/Power Tradeoff Model for Replicated Search Engines". In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. ACM. 13–22. ISBN: 978-1-4503-2351-2. DOI: 10.1145/2556195.2556246.

Freire, A., C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda (2015). "Queuing Theory-based Latency/Power Tradeoff Models for Replicated Search Engines". *J. UCS*. 21(13): 1790–1809. ISSN: 0948-695X. DOI: 10.3217/jucs-021-13-1790.

Freund, Y., R. Iyer, R. E. Schapire, and Y. Singer (2003). "An Efficient Boosting Algorithm for Combining Preferences". *J. Mach. Learn. Res.* 4(Dec.): 933–969. ISSN: 1532-4435.

Friedman, J. H. (2001). "Greedy Function Approximation: a Gradient Boosting Machine". *Ann. Statist.* 29(5): 1189–1232. DOI: 10.1214/aos/1013203451.

Goldstein, J., R. Ramakrishnan, and U. Shaft (1998). "Compressing Relations and Indexes". In: *Proceedings of the 14th International Conference on Data Engineering*. IEEE. 370–379. ISBN: 0-8186-8289-2. DOI: 10.1109/ICDE.1998.655800.

Golomb, S. (1966). "Run-length Encodings". *IEEE Trans. Inf. Theor.* 12(3).

Goodwin, B., M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He (2017). "BitFunnel: Revisiting Signatures for Search". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 605–614. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3080789.

Greenberg, A., J. Hamilton, D. A. Maltz, and P. Patel (2008). “The Cost of a Cloud: Research Problems in Data Center Networks”. *SIGCOMM Comput. Commun. Rev.* 39(1): 68–73. ISSN: 0146-4833. DOI: 10.1145/1496091.1496103.

Harman, D. and G. Candela (1990). “Retrieving Records from a Gigabyte of Text on a Minicomputer using Statistical Ranking”. *J. Am. Soc. Inf. Sc.* 41(8): 581–589. ISSN: 1097-4571.

Hazelwood, K., S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang (2018). “Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective”. In: *Proceeding of the IEEE International Symposium on High Performance Computer Architecture*. IEEE. 10. ISBN: 978-1-5386-3659-6. DOI: 10.1109/HPCA.2018.00059.

He, B. and I. Ounis (2006). “Query Performance Prediction”. *Inf. Syst.* 31(7): 585–594. ISSN: 0306-4379. DOI: 10.1016/j.is.2005.11.003.

He, X., J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. Candela (2014). “Practical Lessons from Predicting Clicks on Ads at Facebook”. In: *Proceedings of the 8th International Workshop on Data Mining for Online Advertising*. ACM. 5:1–5:9. ISBN: 978-1-4503-2999-6. DOI: 10.1145/2648584.2648589.

Heaps, H. S. (1978). *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc. ISBN: 0123357500.

Hennessy, J. L. and D. A. Patterson (2011). *Computer Architecture: A Quantitative Approach (5th ed.)* Morgan Kaufmann Publishers Inc. ISBN: 012383872X.

Järvelin, K. and J. Kekäläinen (2002). “Cumulated Gain-based Evaluation of IR Techniques”. *ACM Trans. Inf. Syst.* 20(4): 422–446. ISSN: 1046-8188. DOI: 10.1145/582415.582418.

Jeon, M., Y. He, S. Elnikety, A. L. Cox, and S. Rixner (2013). “Adaptive Parallelism for Web Search”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 155–168. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465367.

Jeon, M., S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner (2014). “Predictive Parallelization: Taming Tail Latencies in Web Search”. In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 253–262. ISBN: 978-1-4503-2257-7. DOI: 10.1145/2600428.2609572.

Jia, X.-F., A. Trotman, and R. O’Keefe (2010). “Efficient Accumulator Initialisation”. In: *Proceedings of the 15th Australasian Document Computing Symposium*. 44–51.

Jin, X., T. Yang, and X. Tang (2016). “A Comparison of Cache Blocking Methods for Fast Execution of Ensemble-based Score Computation”. In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 629–638. ISBN: 978-1-4503-4069-4. DOI: 10.1145/2911451.2911520.

Joachims, T. (2002). “Optimizing Search Engines Using Clickthrough Data”. In: *Proceedings of the 8th ACM International Conference on Knowledge Discovery and Data Mining*. ACM. 133–142. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775067.

Jonassen, S. and S. E. Bratsberg (2011). “Efficient Compressed Inverted Index Skipping for Disjunctive Text-Queries”. In: *Proceedings of the 33rd European Conference on IR Research*. Springer. 530–542. ISBN: 978-3-642-20161-5. DOI: 10.1007/978-3-642-20161-5_53.

Jones, R., B. Rey, O. Madani, and W. Greiner (2006). “Generating Query Substitutions”. In: *Proceedings of the 15th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 387–396. ISBN: 1-59593-323-9. DOI: 10.1145/1135777.1135835.

Kane, A. and F. W. Tompa (2018). “Split-Lists and Initial Thresholds for WAND-based Search”. In: *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 877–880. ISBN: 978-1-4503-5657-2. DOI: 10.1145/3209978.3210066.

Kaszkiel, M. and J. Zobel (1998). “Term-ordered Query Evaluation Versus Document-ordered Query Evaluation for Large Document Databases”. In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 343–344. ISBN: 1-58113-015-5. DOI: 10.1145/290941.291031.

Kaszkiel, M., J. Zobel, and R. Sacks-Davis (1999). “Efficient Passage Ranking for Document Databases”. *ACM Trans. Inf. Syst.* 17(4): 406–439. ISSN: 1046-8188. DOI: 10.1145/326440.326445.

Kayaaslan, E., B. B. Cambazoglu, R. Blanco, F. P. Junqueira, and C. Aykanat (2011). “Energy-price-driven Query Processing in Multi-center Web Search Engines”. In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 983–992. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2010047.

Kim, S., Y. He, S.-w. Hwang, S. Elnikety, and S. Choi (2015). “Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search”. In: *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*. ACM. 7–16. ISBN: 978-1-4503-3317-7. DOI: 10.1145/2684822.2685289.

Kim, Y. and J. Lin (2018). “Serverless Data Analytics with Flint”. In: *Proceedings of the 11th IEEE International Conference on Cloud Computing*. IEEE. 451–455. ISBN: 9781538672365. DOI: 10.1109/CLOUD.2018.00063.

Kohavi, R. (1994). “Bottom-Up Induction of Oblivious Read-Once Decision Graphs: Strengths and Limitations”. In: *Proceedings of the 12th National Conference on Artificial Intelligence, (AAAI)*. AAAI. 613–618. ISBN: 0-262-61102-3.

Kyriazis, D., C. Doulkeridis, P. Gouvas, R. Jimenez-Peris, A. J. Ferrer, L. Kallipolitis, P. Kranas, G. Kousiouris, C. Macdonald, R. McCreadie, *et al.* (2018). "BigDataStack: A Holistic Data-Driven Stack for Big Data Applications and Operations". In: *Proceedings of the IEEE International Congress on Big Data*. 237–241. ISBN: 978-1-5386-7232-7. DOI: 10.1109/BigDataCongress.2018.00041.

Lacour, P., C. Macdonald, and I. Ounis (2008). "Efficiency Comparison of Document Matching Techniques". In: *Proceedings of the Efficiency Issues in Information Retrieval Workshop at ECIR 2008*. Ed. by R. Blanco and F. Silvestri. Dept of Computing Science, University of Glasgow.

Lafferty, J. and C. Zhai (2001). "Document Language Models, Query Models, and Risk Minimization for Information Retrieval". In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 111–119. ISBN: 1-58113-331-6. DOI: 10.1145/383952.383970.

Langley, P. and S. Sage (1994). "Oblivious Decision Trees and Abstract Cases". In: *Working Notes of the AAAI-94 Workshop on Case-Based Reasoning*. AAAI. 113–117.

Lemire, D. and L. Boytsov (2015). "Decoding Billions of Integers Per Second Through Vectorization". *Softw. Pract. Exper.* 45(1): 1–29. ISSN: 0038-0644. DOI: 10.1002/spe.2203.

Lester, N., A. Moffat, W. Webber, and J. Zobel (2005). "Space-Limited Ranked Query Evaluation Using Adaptive Pruning". In: *Proceedings of the 6th International Conference on Web Information Systems Engineering*. Springer. 470–477. ISBN: 978-3-540-32286-3. DOI: 10.1007/11581062_37.

Lettich, F., C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini (2018). "Parallel Traversal of Large Ensembles of Decision Trees". *IEEE Trans. Par. Dist. Sys.* 14. ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2860982.

Li, P., C. J. C. Burges, and Q. Wu (2007). "McRank: Learning to Rank Using Multiple Classification and Gradient Boosting". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. Curran Associates Inc. 897–904. ISBN: 978-1-60560-352-0.

Lillis, D., F. Toolan, R. Collier, and J. Dunnion (2006). "ProbFuse: A Probabilistic Approach to Data Fusion". In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 139–146. ISBN: 1-59593-369-7. DOI: 10.1145/1148170.1148197.

Lin, J. (2015). "Building a Self-Contained Search Engine in the Browser". In: *Proceedings of the International Conference on The Theory of Information Retrieval*. ACM. 309–312. ISBN: 978-1-4503-3833-2. DOI: 10.1145/2808194.2809478.

Lin, J. and A. Trotman (2015). "Anytime Ranking for Impact-Ordered Indexes". In: *Proceedings of the International Conference on The Theory of Information Retrieval*. ACM. 301–304. ISBN: 978-1-4503-3833-2. DOI: 10.1145/2808194.2809477.

Lin, J. and A. Trotman (2017). “The Role of Index Compression in Score-at-a-Time Query Evaluation”. *Inf. Retr.* 1–22. ISSN: 1573-7659. DOI: 10.1007/s10791-016-9291-5.

Ling, X., W. Deng, C. Gu, H. Zhou, C. Li, and F. Sun (2017). “Model Ensemble for Click Prediction in Bing Search Ads”. In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 689–698. ISBN: 978-1-4503-4914-7. DOI: 10.1145/3041021.3054192.

Liu, T.-Y. (2009). “Learning to Rank for Information Retrieval”. *Found. and Tr. in IR*. 3(3): 225–331.

Liu, X., Z. Zhang, B. Hou, R. J. Stones, G. Wang, and X. Liu (2018). “Index Compression for BitFunnel Query Processing”. In: *Proceedings of the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 921–924. ISBN: 978-1-4503-5657-2. DOI: 10.1145/3209978.3210086.

Lo, D., L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis (2014). “Towards Energy Proportionality for Large-scale Latency-critical Workloads”. In: *Proc. ISCA*. IEEE. 301–312. ISBN: 978-1-4799-4394-4. DOI: 10.1145/2678373.2665718.

Lu, X., A. Moffat, and J. S. Culpepper (2015). “On the Cost of Extracting Proximity Features for Term-Dependency Models”. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM. 293–302. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806467.

Lucarella, D. (1988). “A Document Retrieval System based on Nearest Neighbour Searching”. *J. of Inf. Sc.* 14(1): 25–33.

Lucchese, C., F. M. Nardini, S. Orlando, R. Perego, and N. Tonellotto (2015a). “Speeding Up Document Ranking with Rank-based Features”. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 895–898. ISBN: 978-1-4503-3621-5. DOI: 10.1145/2766462.2767776.

Lucchese, C., F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini (2015b). “QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees”. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 73–82. ISBN: 978-1-4503-3621-5. DOI: 10.1145/2766462.2767733.

Lucchese, C., F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini (2016). “Exploiting CPU SIMD Extensions to Speed-up Document Scoring with Tree Ensembles”. In: *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 833–836. ISBN: 978-1-4503-4069-4. DOI: 10.1145/2911451.2914758.

Macdonald, C., R. McCreadie, and I. Ounis (2018). “Agile Information Retrieval Experimentation with Terrier Notebooks”. In: *Proceedings of DESIRES 2018*.

Macdonald, C., R. McCreadie, R. Santos, and I. Ounis (2012a). "From Puppy to Maturity: Experiences in Developing Terrier". In: *Proceedings of the Open Source Information Retrieval Workshop*.

Macdonald, C., I. Ounis, and N. Tonellotto (2011). "Upper-bound Approximations for Dynamic Pruning". *ACM Trans. Inf. Syst.* 29(4): 17:1–17:28. ISSN: 1046-8188. DOI: 10.1145/2037661.2037662.

Macdonald, C., R. L. Santos, and I. Ounis (2013a). "The Whens and Hows of Learning to Rank for Web Search". *Inf. Retr.* 16(5): 584–628. ISSN: 1386-4564. DOI: 10.1007/s10791-012-9209-9.

Macdonald, C., R. L. Santos, and I. Ounis (2012b). "On the Usefulness of Query Features for Learning to Rank". In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. ACM. 2559–2562. ISBN: 978-1-4503-1156-4. DOI: 10.1145/2396761.2398691.

Macdonald, C., R. L. Santos, I. Ounis, and B. He (2013b). "About Learning Models with Multiple Query-dependent Features". *ACM Trans. Inf. Syst.* 31(3): 11:1–11:39. ISSN: 1046-8188. DOI: 10.1145/2493175.2493176.

Macdonald, C. and N. Tonellotto (2017). "Upper Bound Approximations for BlockMaxWand". In: *Proceedings of the International Conference on the Theory of Information Retrieval*. ACM. 273–276. ISBN: 9781450344906. DOI: 10.1145/3121050.3121094.

Macdonald, C., N. Tonellotto, and I. Ounis (2012c). "Effect of Dynamic Pruning Safety on Learning to Rank Effectiveness". In: *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 1051–1052. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348464.

Macdonald, C., N. Tonellotto, and I. Ounis (2012d). "Learning to Predict Response Times for Online Query Scheduling". In: *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 621–630. ISBN: 978-1-4503-1472-5. DOI: 10.1145/2348283.2348367.

Macdonald, C., N. Tonellotto, and I. Ounis (2017). "Efficient and Effective Selective Query Rewriting with Efficiency Predictions". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 495–504. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3080827.

Mackenzie, J., J. S. Culpepper, R. Blanco, M. Crane, C. L. A. Clarke, and J. Lin (2018). "Query Driven Algorithm Selection in Early Stage Retrieval". In: *Proceedings of the 11th ACM International Conference on Web Search and Data Mining*. ACM. 396–404. ISBN: 978-1-4503-5581-0. DOI: 10.1145/3159652.3159676.

Mackenzie, J., F. Scholer, and J. S. Culpepper (2017). "Early Termination Heuristics for Score-at-a-Time Index Traversal". In: *Proceedings of the 22nd Australasian Document Computing Symposium*. ACM. 8:1–8:8. ISBN: 978-1-4503-6391-4. DOI: 10.1145/3166072.3166073.

Mallia, A., G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini (2017). "Faster BlockMax WAND with Variable-sized Blocks". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 625–634. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3080780.

Manmatha, R., T. Rath, and F. Feng (2001). "Modeling Score Distributions for Combining the Outputs of Search Engines". In: *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 267–275. ISBN: 1-58113-331-6. DOI: 10.1145/383952.384005.

Manning, C. D., P. Raghavan, and H. Schütze (2008). *Introduction to Information Retrieval*. Cambridge University Press. ISBN: 0521865719.

Maron, M. E. and J. L. Kuhns (1960). "On Relevance, Probabilistic Indexing and Information Retrieval". *J. ACM*. 7(3): 216–244. ISSN: 0004-5411. DOI: 10.1145/321033.321035.

McCreadie, R., D. Albakour, J. Manotumruksa, C. Macdonald, and I. Ounis (2016). "Searching the Internet of Things". *Building Blocks for IoT Analytics*: 39–80.

Metzler, D. and W. B. Croft (2005). "A Markov Random Field Model for Term Dependencies". In: *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 472–479. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076115.

Metzler, D. and W. B. Croft (2007). "Linear Feature-based Models for Information Retrieval". *Inf. Retr.* 10(3): 257–274. ISSN: 1573-7659. DOI: 10.1007/s10791-006-9019-z.

Mitra, B., F. Diaz, and N. Craswell (2017). "Learning to Match Using Local and Distributed Representations of Text for Web Search". In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 1291–1299. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052579.

Moffat, A. and J. Zobel (1994). "Fast Ranking in Limited Space". In: *Proc. IEEE Conf. on Data Engineering*. IEEE. 428–437. ISBN: 0-8186-5402-3. DOI: 10.1109/ICDE.1994.283064.

Moffat, A. and J. Zobel (1996). "Self-Indexing Inverted Files for Fast Text Retrieval". *ACM Trans. Inf. Syst.* 14(4): 349–379.

Moffat, A., J. Zobel, and R. Sacks-Davis (1994). "Memory Efficient Ranking". *Inf. Process. Manage.* 30(6): 733–744. ISSN: 0306-4573. DOI: 10.1016/0306-4573(94)90002-7.

Montague, M. and J. A. Aslam (2001). "Relevance Score Normalization for Metasearch". In: *Proceedings of the 10th International Conference on Information and Knowledge Management*. ACM. 427–433. ISBN: 1-58113-436-3. DOI: 10.1145/502585.502657.

Montague, M. and J. A. Aslam (2002). "Condorcet Fusion for Improved Retrieval". In: *Proceedings of the 11th International Conference on Information and Knowledge Management*. ACM. 538–548. ISBN: 1-58113-492-4. DOI: 10.1145/584792.584881.

Nenkova, A. and K. McKeown (2011). "Automatic Summarization". *Found. and Tr. in IR*. 5(2–3): 103–233. ISSN: 1554-0669. DOI: 10.1561/1500000015.

Noreault, T., M. Koll, and M. J. McGill (1977). “Automatic Ranked Output from Boolean Searches in SIRE”. *J. Am. Soc. Inf. Sc.* 28(6): 333–339. ISSN: 1097-4571. DOI: 10.1002/asi.4630280605.

Ottaviano, G., N. Tonellotto, and R. Venturini (2015). “Optimal Space-time Tradeoffs for Inverted Indexes”. In: *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*. ACM. 47–56. ISBN: 978-1-4503-3317-7. DOI: 10.1145/2684822.2685297.

Ottaviano, G. and R. Venturini (2014). “Partitioned Elias-Fano Indexes”. In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 273–282. ISBN: 978-1-4503-2257-7. DOI: 10.1145/2600428.2609615.

Pang, L., Y. Lan, J. Guo, J. Xu, J. Xu, and X. Cheng (2017). “DeepRank: A New Deep Architecture for Relevance Ranking in Information Retrieval”. In: *Proceedings of the 27th ACM on Conference on Information and Knowledge Management*. ACM. 257–266. ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3132914.

Pedersen, J. (2010). “Query Understanding at Bing”. In: *Proceedings of the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Peng, F., N. Ahmed, X. Li, and Y. Lu (2007). “Context Sensitive Stemming for Web Search”. In: *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 639–646. ISBN: 978-1-59593-597-7. DOI: 10.1145/1277741.1277851.

Perry, S. A. and P. Willett (1983). “A Review of the Use of Inverted Files for Best Match Searching in Information Retrieval Systems”. *J. Inf. Sc.* 6(2–3): 59–66.

Persin, M. (1994). “Document Filtering for Fast Ranking”. In: *Proceedings of the 17th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Ed. by W. B. Croft and C. J. van Rijsbergen. ACM. ISBN: 0-387-19889-X.

Persin, M., J. Zobel, and R. Sacks-Davis (1996). “Filtered Document Retrieval with Frequency-sorted Indexes”. *J. Am. Soc. Inf. Sci.* 47(10): 749–764. ISSN: 0002-8231. DOI: 10.1002/(SICI)1097-4571(199610)47:10<749::AID-ASI3>3.3.CO;2-U.

Petri, M., J. S. Culpepper, and A. Moffat (2013). “Exploring the Magic of WAND”. In: *Proceedings of the 18th Australasian Document Computing Symposium*. ACM. 58–65. ISBN: 978-1-4503-2524-0. DOI: 10.1145/2537734.2537744.

Ponte, J. M. and W. B. Croft (1998). “A Language Modeling Approach to Information Retrieval”. In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 275–281. ISBN: 1-58113-015-5. DOI: 10.1145/290941.291008.

Pugh, W. (1990). “Skip Lists: A Probabilistic Alternative to Balanced Trees”. *Commun. ACM*. 33(6): 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977.

Putnam, A., A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger (2014). "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. IEEE. 13–24. ISBN: 978-1-4799-4394-4. DOI: 10.1145/2678373.2665678.

Qin, T., T.-Y. Liu, J. Xu, and H. Li (2010). "LETOR: A Benchmark Collection for Research on Learning to Rank for Information Retrieval". *Inf. Retr.* 13(4): 346–374. ISSN: 1573-7659. DOI: 10.1007/s10791-009-9123-y.

Ramaswamy, V., R. Konow, A. Trotman, J. Degenhardt, and N. Whyte (2017). "Document Reordering is Good, Especially for e-Commerce". In: *Proceedings of Workshop on eCommerce*.

Rice, R. and J. Plaunt (1971). "Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data". *IEEE Trans. Comm. Tech.* 19(6): 889–897. ISSN: 0018-9332. DOI: 10.1109/TCOM.1971.1090789.

Rigutini, L., T. Papini, M. Maggini, and F. Scarselli (2011). "SortNet: Learning to Rank by a Neural Preference Function". *IEEE Trans. Neur. Netw.* 22(9): 1368–1380. ISSN: 1045-9227. DOI: 10.1109/TNN.2011.2160875.

Rijsbergen, C. van (1979). *Information Retrieval (2nd ed.)* Butterworths, London. ISBN: 9780408709293.

Risvik, K. M., Y. Aasheim, and M. Lidal (2003). "Multi-Tier Architecture for Web Search Engines". In: *Proceedings of the 1st Latin American Web Congress*. IEEE. 132–143. ISBN: 0-7695-2058-8. DOI: 10.1109/LAWEB.2003.1250291.

Risvik, K. M., T. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson (2013). "Maguro, a System for Indexing and Searching over Very Large Text Collections". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. ACM. 727–736. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433486.

Robertson, S. E., S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford (1994). "Okapi at TREC-3". In: *Proceedings of the 3rd Text REtrieval Conference*. NIST. 109–126.

Rojas, O., V. Gil-Costa, and M. Marin (2013a). "Distributing Efficiently the Block-Max WAND Algorithm". *Procedia Computer Science*. 18: 120–129. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2013.05.175.

Rojas, O., V. Gil-Costa, and M. Marin (2013b). "Efficient Parallel Block-Max WAND Algorithm". In: *Proceedings of the 19th Euro-Par International Conference*. Springer. 394–405. ISBN: 978-3-642-40047-6. DOI: 10.1007/978-3-642-40047-6_41.

Rossi, C., E. S. de Moura, A. L. Carvalho, and A. S. da Silva (2013). "Fast Document-at-a-time Query Processing Using Two-tier Indexes". In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 183–192. ISBN: 978-1-4503-2034-4. DOI: 10.1145/2484028.2484085.

Sadakane, K. and H. Imai (1999). "Text Retrieval by Using k-word Proximity Search". In: *Proceedings of the International Symposium on Database Applications in Non-Traditional Environments*. IEEE. 183–188. ISBN: 0-7695-0496-5. DOI: 10.1109/DANTE.1999.844958.

Salton, G., A. Wong, and C. S. Yang (1975). "A Vector Space Model for Automatic Indexing". *Commun. ACM*. 18(11): 613–620. ISSN: 0001-0782. DOI: 10.1145/361219.361220.

Salton, G. (1989). *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley. ISBN: 0-201-12227-8.

Scholer, F., H. E. Williams, J. Yiannis, and J. Zobel (2002). "Compression of Inverted Indexes For Fast Query Evaluation". In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 222–229. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564416.

Shan, D., S. Ding, J. He, H. Yan, and X. Li (2012). "Optimized Top-k Processing with Global Page Scores on Block-max Indexes". In: *Proceedings of the 5th ACM International Conference on Web Search and Data Mining*. ACM. 423–432. ISBN: 978-1-4503-0747-5. DOI: 10.1145/2124295.2124346.

Sharp, T. (2008). "Implementing Decision Trees and Forests on a GPU". In: *Proc. Computer Vision*. Springer. 595–608. ISBN: 978-3-540-88693-8. DOI: 10.1007/978-3-540-88693-8_44.

Shieh, W.-Y., T.-F. Chen, J. J.-J. Shann, and C.-P. Chung (2003). "Inverted File Compression Through Document Identifier Reassignment". *Inf. Process. Manage.* 39(1): 117–131. ISSN: 0306-4573. DOI: 10.1016/S0306-4573(02)00020-1.

Silverstein, C., H. Marais, M. Henzinger, and M. Moricz (1999). "Analysis of a Very Large Web Search Engine Query Log". *SIGIR Forum*. 33(1): 6–12. ISSN: 0163-5840. DOI: 10.1145/331403.331405.

Silvestri, F. (2007). "Sorting out the Document Identifier Assignment Problem". In: *Proceedings of the 29th European Conference on IR Research*. Springer. 101–112. ISBN: 978-3-540-71494-1. DOI: 10.1007/978-3-540-71496-5_12.

Silvestri, F., S. Orlando, and R. Perego (2004). "Assigning Identifiers to Documents to Enhance the Clustering Property of Fulltext Indexes". In: *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 305–312. ISBN: 1-58113-881-4. DOI: 10.1145/1008992.1009046.

Smeaton, A. F. and C. J. van Rijsbergen (1981). "The Nearest Neighbour Problem in Information Retrieval: An Algorithm Using Upperbounds". In: *Proceedings of the 4th Annual International ACM SIGIR Conference on Information Storage and Retrieval*. ACM. 83–87. ISBN: 0-89791-052-4. DOI: 10.1145/511754.511767.

Snowdon, D. C., S. Ruocco, and G. Heiser (2005). “Power Management and Dynamic Voltage Scaling: Myths and Facts”. In: *Proc. Workshop on Power Aware Real-time Computing*.

Sorokina, D. and E. Cantú-Paz (2016). “Amazon Search: The Joy of Ranking Products”. In: *Proceedings of the 39th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 459–460. ISBN: 978-1-4503-4069-4. DOI: 10.1145/2911451.2926725.

Stepanov, A. A., A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi (2011). “SIMD-based Decoding of Posting Lists”. In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. ACM. 317–326. ISBN: 978-1-4503-0717-8. DOI: 10.1145/2063576.2063627.

Strohman, T. (2007). “Efficient Processing of Complex Features for Information Retrieval”.

Strohman, T. and W. B. Croft (2006). “Low Latency Index Maintenance in Indri”. In: *Proceedings of the Open Source Information Retrieval Workshop*.

Strohman, T. and W. B. Croft (2007). “Efficient Document Retrieval in Main Memory”. In: *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 175–182. ISBN: 978-1-59593-597-7. DOI: 10.1145/1277741.1277774.

Strohman, T., H. Turtle, and W. B. Croft (2005). “Optimization Strategies for Complex Queries”. In: *Proceedings of the 28th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 219–225. ISBN: 1-59593-034-5. DOI: 10.1145/1076034.1076074.

Tang, X., X. Jin, and T. Yang (2014). “Cache-conscious Runtime Optimization for Ranking Rnsembles”. In: *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 1123–1126. ISBN: 978-1-4503-2257-7. DOI: 10.1145/2600428.2609525.

Tatikonda, S., B. B. Cambazoglu, and F. P. Junqueira (2011). “Posting List Intersection on Multicore Architectures”. In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 963–972. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2010045.

Teymorian, A., O. Frieder, and M. A. Maloof (2013). “Rank-energy Selective Query Forwarding for Distributed Search Systems”. In: *Proceedings of the 2nd ACM International Conference on Information & Knowledge Management*. ACM. 389–398. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505710.

Tonellotto, N., C. Macdonald, and I. Ounis (2010). “Efficient Dynamic Pruning with Proximity Support”. In: *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval*. 31–35.

Tonellotto, N., C. Macdonald, and I. Ounis (2011). "Effect of Different Docid Orderings on Dynamic Pruning Retrieval Strategies". In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 1179–1180. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2010108.

Tonellotto, N., C. Macdonald, and I. Ounis (2013). "Efficient and Effective Retrieval Using Selective Pruning". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. ACM. 63–72. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433407.

Trofimov, I., A. Kornetova, and V. Topinskiy (2012). "Using Boosted Trees for Click-through Rate Prediction for Sponsored Search". In: *Proceedings of the 6th International Workshop on Data Mining for Online Advertising and Internet Economy*. ACM. 2:1–2:6. ISBN: 978-1-4503-1545-6. DOI: 10.1145/2351356.2351358.

Trotman, A. (2014). "Compression, SIMD, and Postings Lists". In: *Proceedings of the 19th Australasian Document Computing Symposium*. ACM. 50:50–50:57. ISBN: 978-1-4503-3000-8. DOI: 10.1145/2682862.2682870.

Trotman, A. and J. Lin (2016). "In Vacuo and In Situ Evaluation of SIMD Codecs". In: *Proceedings of the 21st Australasian Document Computing Symposium*. ACM. 1–8. ISBN: 978-1-4503-4865-2. DOI: 10.1145/3015022.3015023.

Tu, Z., M. Li, and J. Lin (2018). "Pay-Per-Request Deployment of Neural Network Models Using Serverless Architectures". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics*. ACL. 6–10. ISBN: 978-1-948087-28-5. DOI: 10.18653/v1/N18-5002.

Turtle, H. and J. Flood (1995). "Query Evaluation: Strategies and Optimizations". *Inf. Process. Manage.* 31(6): 831–850. ISSN: 0306-4573. DOI: http://dx.doi.org/10.1016/0306-4573(95)00020-H.

Van Essen, B., C. Macaraeg, M. Gokhale, and R. Prenger (2012). "Accelerating a Random Forest Classifier: Multi-core, GP-GPU, or FPGA?" In: *Proceedings of the IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 232–239. ISBN: 978-1-4673-1605-7. DOI: 10.1109/FCCM.2012.47.

Vigna, S. (2013). "Quasi-succinct Indices". In: *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*. ACM. 83–92. ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433409.

Vogt, C. C. and G. W. Cottrell (1998). "Predicting the Performance of Linearly Combined IR Systems". In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 190–196. ISBN: 1-58113-015-5. DOI: 10.1145/290941.290991.

Wang, J., E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu (2013). “The Impact of Solid State Drive on Search Engine Cache Management”. In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 693–702. ISBN: 978-1-4503-2034-4. DOI: 10.1145/2484028.2484046.

Wang, L., J. Lin, and D. Metzler (2010). “Learning to Efficiently Rank”. In: *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 138–145. ISBN: 978-1-4503-0153-4. DOI: 10.1145/1835449.1835475.

Wang, L., J. Lin, and D. Metzler (2011). “A Cascade Ranking Model for Efficient Ranked Retrieval”. In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 105–114. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2009934.

White, R. W. (2016). *Interactions with Search Systems*. Cambridge University Press. ISBN: 9781139525305.

Williams, H. E. and J. Zobel (1999). “Compressing Integers for Fast File Access”. *The Computer Journal*. 42: 193–201.

Witten, I. H., A. Moffat, and T. C. Bell (1999). *Managing Gigabytes (2nd ed.)* Morgan Kaufmann Publishers Inc. ISBN: 1-55860-570-3.

Wong, W. Y. P. and D. L. Lee (1993). “Implementations of Partial Document Ranking Using Inverted Files”. *Inf. Process. Manage.* 29(5): 647–669.

Wu, D., F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu (2010). “Efficient Lists Intersection by CPU-GPU Cooperative Computing”. In: *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. IEEE. 1–8. ISBN: 978-1-4244-6533-0. DOI: 10.1109/IPDPSW.2010.5470886.

Wu, H. and H. Fang (2014). “Analytical Performance Modeling for Top-K Query Processing”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM. 1619–1628. ISBN: 978-1-4503-2598-1. DOI: 10.1145/2661829.2661931.

Xu, J. and H. Li (2007). “AdaRank: A Boosting Algorithm for Information Retrieval”. In: *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 391–398. ISBN: 978-1-59593-597-7. DOI: 10.1145/1277741.1277809.

Yan, H., S. Ding, and T. Suel (2009a). “Compressing Term Positions in Web Indexes”. In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 147–154. ISBN: 978-1-60558-483-6. DOI: 10.1145/1571941.1571969.

Yan, H., S. Ding, and T. Suel (2009b). “Inverted Index Compression and Query Processing with Optimized Document Ordering”. In: *Proceedings of the 18th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 401–410. ISBN: 978-1-60558-487-4. DOI: 10.1145/1526709.1526764.

Yin, D., Y. Hu, J. Tang, T. Daly, M. Zhou, H. Ouyang, J. Chen, C. Kang, H. Deng, C. Nobata, J.-M. Langlois, and Y. Chang (2016). "Ranking Relevance in Yahoo Search". In: *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining*. ACM. 323–332. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939677.

Zobel, J. and A. Moffat (2006). "Inverted Files for Text Search Engines". *ACM Comp. Surv.* 38(2). ISSN: 0360-0300. DOI: 10.1145/1132956.1132959.

Zobel, J., A. Moffat, and K. Ramamohanarao (1998). "Inverted Files Versus Signature Files for Text Indexing". *ACM Trans. Database Syst.* 23(4): 453–490. ISSN: 0362-5915. DOI: 10.1145/296854.277632.

Zukowski, M., S. Heman, N. Nes, and P. Boncz (2006). "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of the 22nd International Conference on Data Engineering*. IEEE. 59–70. ISBN: 0-7695-2570-9. DOI: 10.1109/ICDE.2006.150.