Information Retrieval 24-25

Notes

University of Pisa

M.Sc. in Computer Science

Contents

Chapter 1

Introduction

Information retrieval is the process of finding relevant material of unstructured nature from large collections. The "material" is usually documents, web pages, or multimedia content. Originally, information retrieval was something only a few professionals interacted with. Nowadays, hundreds of millions of people engage with information retrieval systems when they, for example, use a web search engine or search through their email.

The two key aspects considered when evaluating the quality of an IR system are **effectiveness** and **efficiency**. The first refers to the capability of the system to produce a satisfactory result; the second refers to how quickly it does so. To guarantee a certain level of quality, some operations are done offline, such as document indexing, feature processing (e.g., term frequency, metadata), training of a learn-to-rank model to produce the order in which documents will be shown to the user, and so on. Still, many operations must be done on-line, such as query expansion and processing, index and feature lookup, and usage of the ranking funcion. If we consider the example of a search engine (SE), we are used to get back a response in a very short amount of time, despite the fact that in order to find the collection of documents presented to us, a lot of different operations must have been performed (i.e., the system must be efficient). Additionally, we also expect that those documents are the most relevant ones found in the collection, and that they are presented in the order of relevancy (the system must be effective). If those two ideas do not hold, we're unlikely to actually use the system for an extended amount of time.

The following chapters will go in detail about the different components of IR systems, and each will focus on how effectiveness and efficiency can be guaranteed. The key aspects that will be considered are:

- **Language properties:** how does language influence retrieval? What does it mean to retrieve a piece of text? How are documents scored and presented?

- **Auxiliary data structures:** e.g., inverted indexes;
- **Query processing:** how are queries expanded from the form provided by the user into one which can be "read" by the system?
- **Data storage and compression:** how can data be compressed efficiently?
- **Learning-to-rank models:** how is machine learning used in an IR system to produce a ranking (based on available ranked data)?
- **Neural IR:** how can Deep Learning and specifically Large Language Models be used in IR?

Chapter 2

Evaluation

To evaluate the quality of a IR system, say a SE, we may ask questions such as: how fast does it index a collection, how fast does it search (efficiency)? Or, does it recommend good related pages/products to buy to the user (effectiveness)? However, these questions alone do not provide any objective information about the intrinsic quality of the SE. We could say that a SE is "good" if it makes its users happy; but to measure this happiness, we can use different definitions: for example, how many times a search result is clicked, how long users stay on the same webpage, how often they return to use the SE. Since happiness by itself is impossible to measure, a commonly used proxy is **relevance** of search results.

2.1 Relevance

In order to measure relevance, three elements are needed:

- A benchmark document collection;
- A benchmark suite of queries;
- An assessment of either **relevant** or **non-relevant** for each query and document.

To construct the benchmark, we would have to analyze each possible pair of query and document and assign a relevance to it. Relevance assessment can be binary (relevant/not relevant), or multi-valued (0, 1, 2, 3 ...) for more nuance. Obviously, since assigning a relevance value to each query-document pair in a collection is way too expensive, a subset of the documents is used instead.

Assigning relevance must be done externally by humans. Some companies use crowdsourcing platforms (e.g., Amazon Mechanical Turk) to present pairs to low cost, not

highly qualified workers. This solution is cheap, but the outcome may not be as good as one produced by professionals.

But how are the queries defined? They must be relevant and suitable to the documents in the collection, and must be representative of user needs (i.e., they should resemble a normal query done by a person). One way to find them is to sample directly from existing query logs of the SE, if available. For classical, non-Web IR systems, these query logs may be nearly empty, as the query rate tends to be slow. In this case, experts may handcraft "user needs" and associated queries. Among popular public test collections is **TREC** (**Text REtrieval Conference**), where focus areas are called **tracks**; each track has a motivating use case, usually an abstraction of a user task. In practice, TREC consists of:

- A set of documents;
- A set of information needs (called **topics**);
- Relevance judgements that indicate which documents should be retrieved by which topics.

The result of a retrieval system executing a task on a test collection is called **run**. The technique first used to select the sample of documents to present to a human judge is **pooling**: the top results for a set of runs are combined to form a pool and only those documents are judged. Since this method automatically assumes that all unpooled documents are not relevant (so they remain unjudged), alternative methods have been investigated by TREC tracks to obtain judgements that support fair evaluation.

Note that TREC does not contain any query, and only generic user needs. Participants are free to define (manually or automatically) actual queries for their specific IR system.

2.1.1 Measures

Evaluation of Unranked Retrieval Sets

Precision and **recall** are binary assessments commonly used to evaluate the effectiveness of an IR system. They are defined on the basis of a set of counts, described by the table below.

	Retrieved	Not retrieved
Relevant	TP	FN
Non-relevant	FP	TN

The two measures are then defined as:

- **Precision:** fraction of retrieved documents that are relevant.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall:** fraction of relevant documents that are retrieved.

$$Recall = \frac{TP}{TP + FN}$$

The **F-Measure** (or **F-Score**) is another metric which condenses both precision and recall; it's calculated as the harmonic mean of the two:

$$F = \frac{1}{0.5Prec. + 0.5Rec.} = 2\frac{Prec. \cdot Rec.}{Prec. + Rec.}$$

The harmonic mean is alway less or equal than the arithmetic/geometric mean: if the two values are very different, the harmonic mean is closer to their minimum. A weighted variant also exists; let α be the weight assigned to precision and β the weight assigned to recall, such that $\alpha = 1/(1 + \beta^2)$, **weighted F-Measure** is calculated as:

$$F_\beta = (\beta^2 + 1)\frac{Prec. \cdot Rec.}{\beta^2 Prec. + Rec.}$$

Evaluation of Ranked Retrieval Results

Precision, recall, and F-score are set-based methods, meaning that they are computed using an unordered set of documents. They can be extended to evaluate the ranked retrieval results returned by search engines. Some of these measures are **Mean Average Precision** (**MAP**), **Precision@K** (**P@K**), **Mean Reciprocal Rank** (**MRR**) for binary relevance, and **Normalized Discounted Cumulative Gain** (**NDCG**) for multiple levels of relevance.

Mean Average Precision Consider a set of documents in a run, classified as relevant or not relevant by the SE. Consider the indexes $K_1, \ldots, K_R$ at which recall increases (i.e., the indexes of the actually relevant documents in that run). Precision is calculated at each K_i, considering only the documents with lower indexes, and the average across all K_i is calculated at the end. MAP is then calculated as the mean of the averages across multiple queries/rankings, as a measure of the whole system given that benchmark.

MAP is a macro-averaging measure; each query counts equally, even if only few documents are relevant for certain queries and many are relevant for other queries.

Precision@K MAP takes into account all documents returned by the query. However, especially in Web searches, the number of retrieved documents may be unknown or very high; what truly matters is how any good results are found in the first few pages. To calculate Precision@K, we set a rank threshold K, we compute the number of relevant documents among the top K ranking ones, and calculate P@K as:

$$P@K = \frac{\sum relevant\ and\ retrieved\ documents\ in\ top\text{-}K}{K}$$

i.e., it is the fraction of relevant documents across the top-K retrieved ones (which are assumed to be relevant by the system). In a similar fashion, we also calculate **Recall@K** as the fraction of relevant documents that are found among the top-K:

$$R@K = \frac{\sum relevant\ and\ retrieved\ documents\ in\ top\text{-}K}{\#\ of\ relevant\ documents}$$

P@K by itself does not average very well over a set of queries, since the number of relevant documents for a query affects the result. Anyway, it (along with **MAP@K**) are largely used for Web searches and recommender systems.

Mean Reciprocal Rank Suppose there is only a single relevant document for a given query. A way to approximate the rank of the correct answer could be to consider the search duration for the user: if its ranking is low, he/she takes a long time to find the answer; if it is ranked high, he/she finds it quickly.

Consider the rank position $rank_i$ of the first relevant document returned by a query $q_i \in Q$. The **Reciprocal Rank** is calculated as:

$$RR = \frac{1}{rank_i}$$

MRR is the mean of the RRs across multiple queries.

Evaluation of Non-Binary Relevance

A popular measure used to evaluate web searches with non-binary relevance is **cumulative gain**, and in particular **Normalized Discounted Cumulative Gain** (**NDCG**). The idea is that the lower the rank of a relevant document is, the less it is useful for the user, since it is less likely to be visited.

Discounted Cumulative Gain uses graded relevance (or **gain**) as a measure of usefulness; it is accumulated starting at the top of the ranking and may also be discounted (= reduced) at lower ranks. The typical discount is $\frac{1}{\log(rank)}$. Let r_i be the relevancy of

the document; then, DCG is calculated as:

$$DCG = r_1 + \frac{r_2}{\log_2 2} + \frac{r_3}{\log_2 3} + \cdots + \frac{r_n}{\log_2 n} = \\ = r_1 + \sum_{i=2}^{n} \frac{r_i}{\log_2 i}$$

This case uses base 2 for the logarithm, but any other base can be used as well. As for the other measures, it can be calculated only for the p top ranking documents instead of the whole set of retrieved ones.

An alternative formulation makes it so that high relevance judgements become much more important:

$$DCG = \sum_{i=1}^{n} \frac{2^{r_i} - 1}{\log_2(1+i)}$$

This variant is used by some web search companies.

NDCG is the normalized version of DCG, and is calculated as the ratio between the DGC of a response and the ideal DGC of a perfect ranking; the perfect ranking would be one that first returns all the docuents with the highest relevance level, then the next highest relevance level, and so on.

$$NDCG = \frac{DCG}{Ideal\ DGC}$$

NDCG takes values between 0 and 1.

Chapter 3

Efficient Algorithms for Modern CPUs

Modern IR systems must manage billions of documents and queries, so we need efficient algorithms that can handle such amounts of data, as well as scale across global infrastructures. Efficiency does not only mean a better user experience, but also a reduction in costs, both for computing and cooling systems. Additionally, efficient systems consume less energy, and therefore are more environmentally sustainable.

Computers use several layers of cache on their chips to speed up RAM and disk access time. When an instruction or a block of data must be read, it is also stored accordingly into the cache(s), since they are faster to access. The smaller the cache, the faster the access time. Ideally, programs should take in consideration how cache layers are used and exploit temporal and spatial locality.

- **Temporal locality** states that when a block of data has been accessed, it is likely to be accessed again very soon. Cache replacement policies such as LRU make sure to keep the data that is most likely to be needed.
- **Spatial locality** states that when a block of data has been accessed, it is likely that nearby memory locations will also be accessed in the near future.

  Normally, when a location is accessed, a larger chunk of memory is read (a cache line of 64 bytes). **Hardware prefetchers** can observe the behaviour of a program and prefetch data if repetitive patterns of cache misses appear.

3.1 Parallelism

Another interesting thing to consider is **parallelism**. Parallelism can speed up a CPU through:

- **Pipelining**, which overlaps the execution of multiple instructions so that different parts of the CPU are kept busy at the same time;
- **Superscalar processors**, which have multiple execution units that process independent operations simultaneously;
- **SIMD**, which are a special kind of instructions executing the same operation on more data at the same time.

Pipelining When an instruction is executed, it actually goes through multiple stages on the CPU. The most simple pipelining is a 5-stage one: fetch, decode, execute, load/store, write. The time needed to move an instruction from one stage to the other defines the **clock time**, so it is chosen to accomodate the longest possible operation (usually memory access).

Modern high-performance CPUs have multiple pipeline stages, usually 10-20, but may be more. This means that the latency to execute something simple like an `add` operation may need up to 20 or more cycles. **Latency** is the total time that an operation passes in the pipeline, while **throughput** of a CPU is the number of instructions that are completed and exit the pipeline per unit of time.

Pipeline hazards are situations where the next instruction cannot go forward in the pipeline in the next clock cycle. This can happen because of:

- **Structural hazards**, when one or more instructions must wait because another one further in the pipeline is using a component. These hazards are unavoidable.
- **Data hazards**, when an instruction must wait for an operand to be computed from a previous step. They can be avoided by restructuring computation;
- **Control hazards**, when the CPU cannot tell which branch in an `if-else` statement it must choose. Normally, it will choose randomly and keep loading instructions into the pipeline, and, if the choice turns out to be wrong, the pipeline will be flushed to accomodate the correct branch (thus wasting some cycles).

Regarding the last type, thankfully CPUs have **branch predictors** capable of guessing which branch is the more likely to be picked by observing past behaviour. The branch predictor is capable of noticing particular patterns, such as a condition holding true/false for several loop iterations, or a condition alternating between true/false between successive instructions.

Superscalar Processing In superscalar processing, multiple instructions are dispatched to do different execution units on the core (each core has several specialized ALUs). This type of parallelism is also called **instruction-level parallelism**.

Single Instruction, Multiple Data SIMD instructions operate n special registers that hold 128, 256, or even 512 bits. Data in registers is divided into blocks of 8, 16, 32, or 64 bits.

SIMD instructions can be used in two ways: either through **auto-vectorization**, meaning that the compiler automatically converts scalar operations into SIMD ones, or they are explicitly used by the programmer in the code. In the latter case, there is an higher level of control over which operations are optimized (since the automatic conversion done by the compiler can only optimize simpler instructions), but obviously requires detailed hardware knowledge.

Chapter 4

Natural Language

Natural language is the language used by humans to communicate with each other. Languages are defined on the basis of thousands of words, a complex syntax, and a mostly compositional semantic. Language is also often ambiguous; the same word may have different meanings depending on the context of the sentence, or the same sentence may be interpreted in different ways.

Natural Language Understanding has the aim of building machines capable of receiving and giving information using natural language, like a human would. Natural language processing is said to be an "AI-complete" problem, meaning that it is a problem only solvable using AI, and that if we were to find a solution, then we'd have a solution for any other problem. In practice, natural language processing is used in many applications, such as chatbots, search engines, machine translation, and personal assistants (such as Alexa by Amazon, or Google Home by Google). It can be used with different data types: tabular data (to construct and interpret search queries), graphs (to represent the content of each node and link, for example in a social media graph), and images/video data (to describe the content of the image/video and to retrieve or generate similar images/videos).

Information Retrieval is strictly connected and overlapping with the fields of Natural Language Processing and Machine Learning:

- NLP methods are often built on top of IR or ML methods;
- ML uses IR measures to define the goals of the learned models, and assumes language can be manipulated via NLP.

4.1 NLP Pipelining

A **processing pipeline** is a sequence of preprocessing steps aimed at transforming the raw text input into a format that can be effectively used as input to the chosen machine learning model(s). Some of the key steps in the pipeline are:

- **Tokenization**: it identifies the words (tokens) in the text. Popular libraries provide "language aware" tokenization, i.e., it isolates tokens differently depending on the language or context of the text. After this step, a **vocabulary** of the terms used can be constructed.
- **Sentence splitting**: it isolates whole sentences from the text, so that they can be analyzed individually. This is not always an easy task, as punctuation marks can often be used for uses other than sentence separation (e.g., in acronyms, numbers, initials, etc.).
- **Stemming and lemmatization**: both aim at reducing words to their roots; stemming does it by applying a set of language-dependent transformation rules to find the stem of a word, while lemmatization actually tries to find the root of the word in the vocabulary. The difference between the two can be shown with an example: the word "cars" is both stemmed and lemmatized to "car"; the word "was" may be stemmed to "wa", but lemmatized to "be".

Raw text can be transformed in different formats depending on the task to be solved. Common models are bag-of-words and n-grams.

Bag-of-words represents a text as the list of unique words appearing in it. It loses any information about word frequency and order, requiring external data structures to store this information; on the other hand, it is easy to implement and calculate. The set of all distinct extracted words can be called *dictionary*, *vocabulary*, or **feature set/space**, since each word becomes a feature of the new representation.

N-grams features are sequences of n words which capture word order. They do not correspond to a specific token in the text, and are instead obtained as a combination of them. For example, in the text:

"the quick brown fox jumps over the lazy dog"

we can find 2-grams such as:

'the quick", 'quick brown', 'brown fox', ... , 'the lazy', 'lazy dog'

Different libraries have specific formats to identify n-grams.

4.2 Zipf's Law

Zipf's Law is an empirical law used to model the frequency of words in a text. It is based in the observation that the most common word in a text is usually twice as frequent as the second most common one, three times as frequent as the third most common, and so on. Specifically, the law states that the frequency of a word is proportional to the inverse of the rank:

$$frequency = \frac{1}{(rank + b)^a}$$

where $a \approx 1$, $b \approx 2.7$.

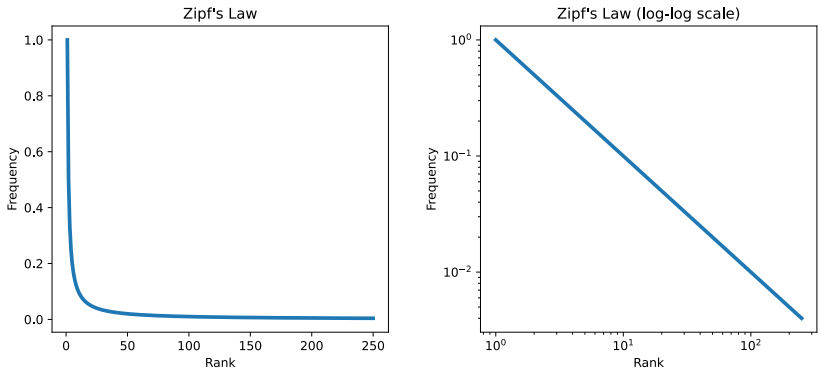

Figure 4.1: Plots showing the frequency of words (in the English language) according to Zipf's Law.

The **principle of least effort** is a theory that states that animals, people, and even well-designed machines naturally choose the path of least effort, meaning the one that requires the least amount of work to reach a goal. In the case of human communication, both speaker and listener in a conversation will abide by this principle. The speaker tends to use a small vocabulary of common words, while the listener tends to prefer longer, rarer words. Zipf's Law can be seen as the result to the compromise between the two.

Stopwords are the most common words in a language. In the English lanaguage, stopwords include "the", "a", "to". They can be removed from the text without losing too much information. Different libraries and tools provide their own list of stopwords depending on the application; for example, MySQL specifies words such as "appreciate" or "unfortunately", since it's used for sentiment analysis.

When it comes to rare words, it is not necessarily true that uncommon words are the most informative or useful. If a word is so rare that it appears very few times in very few documents, it may actually be of little help for future retrieval: it could be

a typo, or a sort of artificial identifier associated to the document (e.g., a slug in a url). Removing these rare words can help make it faster to process indexed data, and requires less space.

4.3 Vector Space Model

Words can be represented as $|F|$-dimensional vectors obtained through **one-hot encoding**, where F is the set of distinct features (words, n-grams, etc.) in the collection. The relevance of a feature f in the document d is indicated as $w_{f,d}$.

A document can then be represented as the weighted sum of all the vectors of its features:

$$v(d) = \sum_{f \in d} w_{f,d} v(f)$$

These vector representations are usually sparse (most of their terms are equal to 0). Weights can be assigned in different ways. Common methods are binary weights (bag-of-words), term frequency (tf), and tf-idf.

To find matches between a query and a document, we can compare the terms appearing in both: using a simple boolean operator (AND/OR), we can check wheter a certain document contains only/all the terms in the query and return them to the user. This is a very simple methodology, but it does not produce a ranking and does not take into account the frequency of words.

In **ranked retrieval**, the system reorders the (top k) documents in the collection for a given query. Instead of defining binary queries with a specific language, natural language is used instead. To rank documents, we need some way to assign a **score** to each document given a query that measures how well they match.

A possible scoring is given by **Jaccard's coefficient**:

$$jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where A and B are sets (in our case, the query and the document). This coefficient is always a value between 0 and 1, where 0 means no overlap and 1 means perfect overlap. While a better alternative than basic binary scoring, it is based on the assumpion that its operators are sets, so it does not consider term frequency to calculate the score. The more frequent is the term in the document, the higher the score; the rarer the term in a collection, the more informative it is (with the exceptions mentioned above).

4.3.1 TF-IDF Scoring

We introduce two data structures to then define more sophisticated scoring measures: the incidence matrix and the count matrix.
The **incidence matrix** is used to represent the presence of absence of terms in documents. Each row corresponds to a word, and each column corresponds to a document.

	d_1	d_2	d_3	d_4
w_1	1	0	1	0
w_2	0	1	0	1
w_3	1	1	0	0
w_4	0	0	1	1

Table 4.1: Example of an incidence matrix.

The **count matrix** keeps track of the number of occurrences of a term in a document, so each column of the matrix is a count vector, and each count vector is a multiset.

	d_1	d_2	d_3	d_4
w_1	32	0	5	0
w_2	0	15	0	13
w_3	2	1	0	0
w_4	0	0	10	2

Table 4.2: Example of a count matrix.

Note that this representation still does not consider the order in which words appear in the documents.

The **term frequency** $tf_{t,d}$ of a term t in a document d is simply the number of times t appears in d. Raw term frequency cannot be used as is, however, since relevancy is not linearly proportional to it. A document with 10 occurrences of a term is more relevant than a document with only 1 occurrence, but it is not 10 times more relevant. **Log-frequency weight** is a common way to normalize term frequency:

$$w_{t,d} = \begin{cases} 1 + \log_{10}(tf_{t,d}) & tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The score of a document-query pair is then calculated by summing the weights of all the terms appearing in both:

$$score(q, d) = \sum_{t \in q \cap d} w_{t,d}$$

The score is always 0 if none of the terms of the query appear in the document.

However, we previously mentioned that informativeness also depends on how frequent the word is in the entire collection. If a term is globally rare, documents containing that word will be more relevant for queries asking for it. To capture this aspect, **document frequency** df_t is used: it is defined as the number of documents containing term t. Since df_t measures the inverse of the informativeness of a word, **inverse document frequency** (idf_t) is used instead:

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$

where N is the number of documents in the collection. The logaritm is used for the same reason as for term frequency. Inverse document frequency by itself cannot rank documents in one-term queries, since *idf* is always either 0 if the document does not contain the term, or a positive value if it does (obtained by the formula above). For multi-term queries, scoring is given by:

$$score(q, d) = \sum_{t \in q \cap d} idf_t$$

In this formula, the role of the document is to filter out the terms which contribute to its own ranking.

These two measures are combined into what's known as **tf-idf weighting**, which is the product between them:

$$tf\text{-}idf_{t,d} = w_{t,d} \cdot idf_t$$

This is the most common weighting scheme in Information Retrieval. Scoring of a document-query pair is calculated as:

$$score(q, d) = \sum_{t \in q \cap d} tf\text{-}idf_{t,d}$$

There are many variants which differ in the way the components are weighted, or the way single terms in the query are weighted as well.
A frequently used variant is **Best Match 25** (**BM25**), which calculates the score as:

$$score(q, d) = \sum_{t \in q \cap d} idf_t \cdot \frac{tf_{t,d} \cdot (k_1 + 1)}{tf_{t,d} + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})}$$

$$idf_t = \ln\left(\frac{N - df_t + 0.5}{df_t + 0.5} + 1\right)$$

where $|D|$ is the length of the document (how many words it contains), *avgdl* is the average length of the documents in the collection, and b, k_1 are hyperparameters; $b \in [0, 1]$ and $k_1 \in [1.2, 2.0]$.

4.3.2 Vector Space Similarity Scoring

Since *tf-idf* can be calculated for each term-document pair, we can construct an additional data structure, called **weight matrix**, which stores all these values. Each document (which is a column in the matrix) can be then thought of as a vector representing a point in $\mathbb{R}^{|F|}$.

The scoring introduced above, however, still presents a problem. Ideally, we want to consider not only the relevancy of single words, but also how relevant a document is in the entire collection. If a document has many rare terms, it should rank higher than another document which has less rare terms. To solve this issue, we can rank documents according to how similar/distant they are from the query (which can be represented as a $|F|$-dimensional vector as well).

A first similarity measure is **Euclidean distance**. It is simple to calculate, and does return a value that is greater the more distant the two vectors are in the space considered. However, Euclidean distance can end up being way too high for vectors of different lenghts (i.e., unnormalized), even if they have the same orientation: for example, consider a document d and append it to itself, obtaining d'. Even though the two documents have the same content (and will have the same direction in the vector space), their magnitude is different, with d''s being exacly double that of d. The key idea is then to rank documents depending on the angle forming between them.

Ranking documents in decreasing order of the angle between query and document is equivalent to ranking them in increasing order of the cosine of the angle. This measure is called **cosine similarity**:

$$\cos(\theta) = \frac{q \cdot d}{\|q\| \cdot \|d\|} = \frac{\sum_{i=1}^{|F|} q_i d_i}{\sqrt{\sum_{i=1}^{|F|} q_i^2} \cdot \sqrt{\sum_{i=1}^{|F|} d_i^2}}$$

Note that here vectors are length-normalized by dividing each for its L2 norm. In the previous example, d and d' would end up having the same lenght after being normalized, so their cosine similarity would be 1 (the maximum possible value).

Different search engines can use different weighting and different normalizations to calculate *tf-idf* and vector similarity. In the **SMART** notation, each method is associated to a different letter (e.g., n for natural/no weighting, l for logarithm, b for boolean, etc.), and the combination used by the engine is specified by a string of six letters: *ddd.qqq*. Different weightings can be used for queries and documents. A standard scheme is ***lnc.ltc***: logarithmic *tf*, no *idf*, cosine similarity normalization for documents, and logarithmic *tf* *idf*, and cosine similarity normalization for queries.

Chapter 5

Indexing

When a user query is submitted to a IR system, the terms in the query must be compared to those contained in the documents in the collection in order to retrieve the relevant ones. If these systems were to use the vector/matrix representation of queries and documents as they are, the complexity of the retrieval would be too high to provide a reasonable response time: we'd need to compute the similarity (i.e., product) between the query vector and all the document vectors in the collection, which are many and with a high number of dimensions. However, these representations tend to be very sparse, meaning that a lot of information could be summarized in a more compact form.

Inverted indexes are the most effective data structures used in IR systems for efficient query processing. They consist of a collection of lists (called **posting lists**), each corresponding to a term in the collection and containing information about the documents in which the term appears in. Each term in the index is also accompanied by its document frequency and its collection frequency; each document ID in the posting list is accompanied by the term frequency in that document. Document IDs are kept sorted in increasing order.

Inverted indexes are often stored in main memory to reduce access time; since they can be very big (the dictionary usually contains a huge number of distinct terms, and the collection holds a huge number of documents), lossless compression algorithms are used to reduce their size.

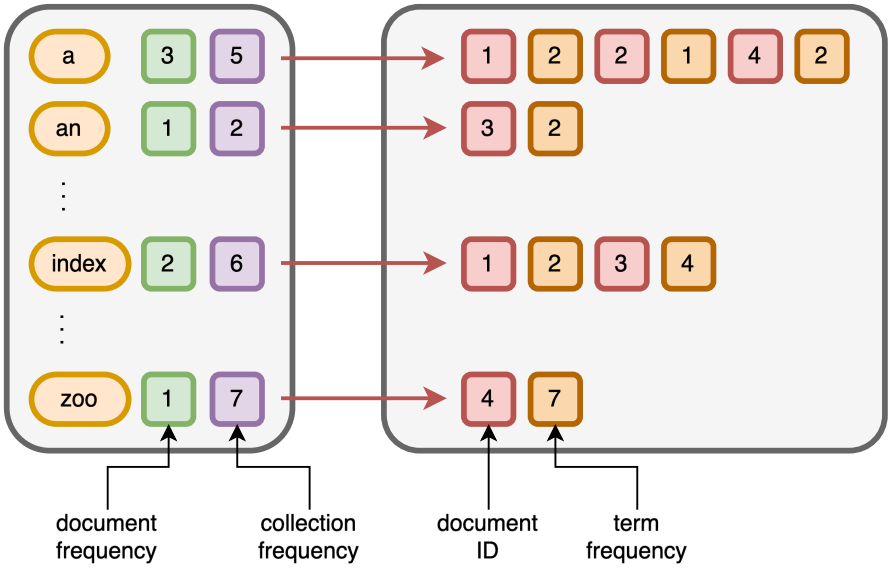

Figure 5.1: Example of inverted index.

5.1 Query Processing

This section will illustrate the fundamental query processing strategies for Boolean retrieval (when an exact answer is required), and ranked retrieval (when a ranked list of scored documents is required). We will assume that the query has already been pre-processed. The two classical query processing strategy categories are **term-at-a-time** (**TAAT**), and **document-at-a-time** (**DAAT**).

TAAT In TAAT, posting lists are read one query term at a time, and accumulators are used to keep track of the documents that satisfy the query; these accumulators may be simple counters or sum up the document score. Accumulators are normally stored in a direct access table or hash table: the first data structure uses the document ID as the key to access the accumulator, while the second uses an hash key calculated from the document ID. If a query is in conjunctive (AND) mode, only the IDs with accumulators equal to the number of query terms are returned; if it is in disjunctive (OR) mode, all the IDs with non-zero accumulators are returned. These approaches are very simple to implement but tend to produce a lot of cache misses because of the large number of accumulators needed. Also, there is no way to skip document IDs for selective queries.

DAAT In DAAT, the posting lists of all the query terms are scanned in parallel, calculating the document score when it is seen in one or more posting lists. Depending

on the mode of the query, a different strategy is used:

- For OR queries, the union of all the posting lists must be computed. The lists are scanned in parallel and merged with the same approach used in the merge phase of a K-way Merge Sort.
- For AND queries, the intersection of all posting lists must be computed. Similarly to before, the lists are scanned in parallel, adding an ID to the result only if all the pointers are pointing to that same ID, otherwise advancing the pointer(s) with the lowest ID without adding anything to the result.

5.1.1 Efficient Document ID Searching

The main operations on posting lists are:

- `first`(): places the pointer to the first docID
- `docID`(): returns the current docID
- `position`(): returns the position of the pointer
- `next`(): moves the pointer to the next docID
- `nextGEQ`(d): moves the pointer to the next docID that is greater or equal than d

This last operation allows us to skip some document IDs, without having to scan all the posting list; this is useful when the query to be processed is in AND mode. `NextGEQ` can be implemented in different ways. Common choices are Binary Exponential Search and skip pointers.

Binary exponential search is similar to basic binary search, but uses a step size that doubles (i.e., increases exponentially) at each iteration, starting from stepsize 1. Assuming there are t `NextGeq` calls on the same list, and n elements in the list, the algorithm runs in $\Theta(\log(\frac{n}{t}))$ time; this estimate assumes that each call operates on a sub-list of size $\frac{n}{t}$. In practice, it is not very efficient because of really small or really big skips. It also cannot be used with most compression algorithms, because they don't allow random access.

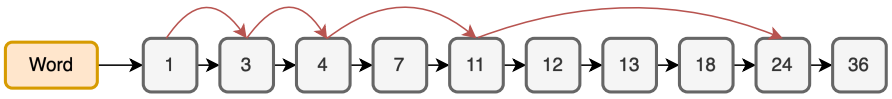

Figure 5.2: Example of exponential binary search.

Skip pointers can be placed to skip k elements at once: each pointer is labeled with the

value it points to. This way, while the searching algorithm encounters pointers with labels smaller or equal than the current ID, it can use the pointers instead of moving one element at a time. This algorithm is usable with compression (since entire blocks of ajdacent IDs can be compressed together).

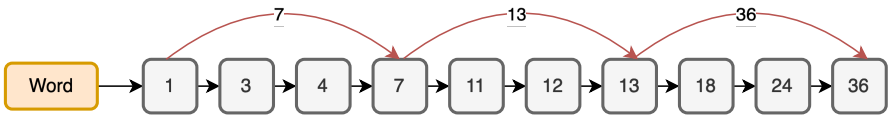

Figure 5.3: Example of skip pointers.

In DAAT, `nextGEQ` is used to skip IDs during the parallel reading of the posting lists. In TAAT, posting lists are intersected from shortest to longest: this is done to make sure each step produces the smallest possible intermediate result.

5.1.2 Positional Indexes

Simple conjunctive/disjuntive queries are actually quite rare in practice, since users tend not to use the dedicated functionalities provided by the IR system. Many search engines provide advanced query language supporting operators such as **phrase queries**: for example, if a user searched for "information retrieval", the result should contain documents where those two words appear close to each other (as a phrase), and not just separately as if they were semantically disconnected.

To support such queries, the inverted index must be expanded to also include the positions of all occurrences of a term within a document. If a set of words constitutes a phrase, then, fixed the document, we need to look for an occurrence of the first word in position p, an occurrence of the second in position $p + 1$, and so on. This way, instead of computing the frequency of single terms, we consider the frequency of the entire set.

5.2 Dynamic Pruning

Since users are only interested in the top few pages in the result of a query, it is not necessary to compute the score of all documents whose ID is contained in the posting lists of the query terms. Strategies for dynamic pruning of the IDs exist for both TAAT and DAAT, and both conjunctive and disjunctive queries. The basic idea behind these strategies is to keep some additional information about the scores calculated so far in order to decide whether to consider or skip certain elements. The two approaches shown next (**WAND** and **MaxScore**) store the top-k IDs in a min-heap and uses the current

minimum as a threshold to choose whether to add a document to the heap or to skip it; as a way to prune the search space, an upper bound of the actual score is used.

Min-heaps A **heap** is a tree-based data structure that satisfies the following property: *if A is a parent of B, then the key of A is ordered w.r.t. the key of B according to a certain criterion.* A **min-heap** is a heap where the key of each node is always less or equal than those of its children, and the minimum key belongs to the root node.

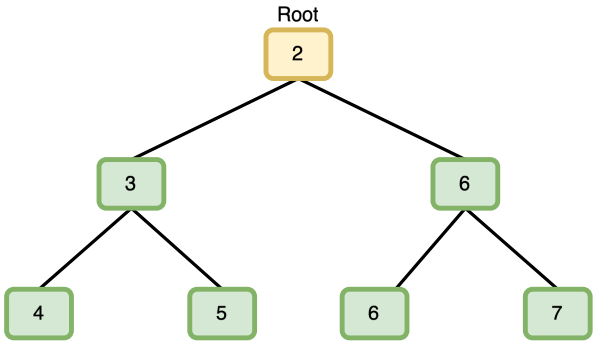

Figure 5.4: Example of min-heap.

Since the root represents the k^{th} highest ranking document found so far, its key can be used as a **threshold** (indicated by the letter τ) to decide if a document should be added to the heap (popping the root and readjusting the heap as necessary), or skipped.

WAND WAND (**Weak AND**) is a pruning strategy that is guaranteed to compute the exact top-k results while skipping many postings in the process. The key idea is to store, for each posting list, the corresponding upper bound of the score, equal to the highest score in the entire list. Before we even calculate the real score of the document, we compare the sum of these upper bounds with the current threshold. If the sum is smaller than τ, the real sum is not computed, and pointers in the posting lists are moved forward; otherwise, the real score is calculated, and placed in the correct spot in the min-heap.

MaxScore MaxScore uses the current threshold to split the posting lists into **essential** ones and **non-essential** ones. The lists are first sorted by their upper bounds, and then scanned in order; the first list whose upper bound is sufficiently high that, if summed to any lower bound, would exceed τ is considered essential, as well as all the others above it. The remaining lists, which have upper bounds so low that even if summed all together result in a value lower than τ are considered non-essential. During

query processing, a document can only be returned as a top-k result if at least one of its terms is in an essential list.

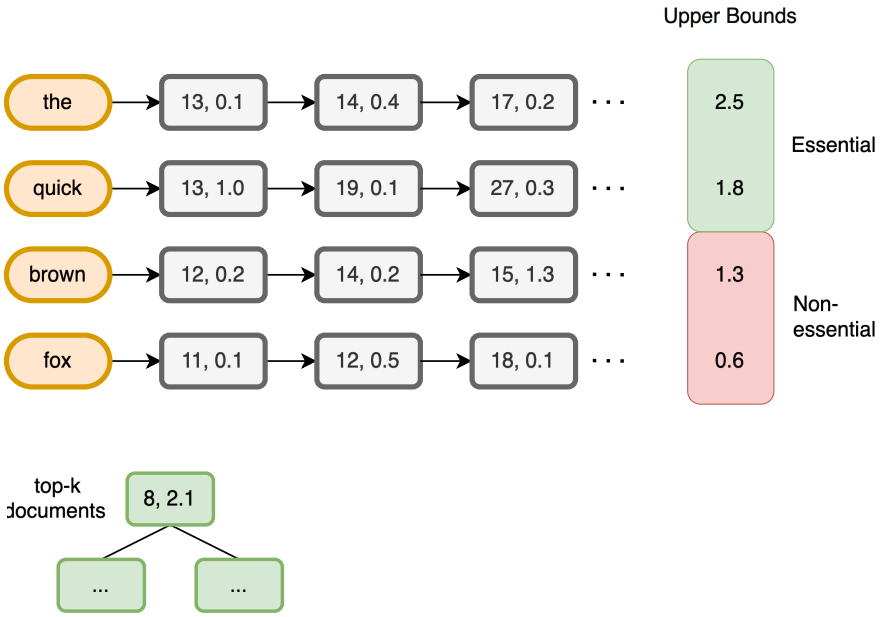

Figure 5.5: Example of splitting of the posting lists in MaxScore.

Chapter 6

Data Compression

Modern database systems are distributed across multiple physical nodes in a network. This distribution is necessary to handle the massive document collections held by search engines and to improve query processing performances. **Sharding** is a technique used to split data into smaller chunks, to be then assigned to the nodes of a network. Sharding ensures not only scalability and performance, but also fault tolerance, since shards are duplicated across different nodes to prevent data loss or slow-downs in case of node failures.

Indexes are also sharded. The assignment of shards to nodes can be represented as a matrix, where each column is a shard, and each row is a node. When a query must be processed, a component called **broker** submits the query to different rows (machines) trying to balance the load, considering which nodes have which shards. The bigger the index is, the more shards are needed; the more users are querying the system, the more replicas are needed. In both cases, it is useful to increase the number of machines.

A way to reduce the number of shards needed to partition the index is to use **compression**. Compression algorithms are either lossless (i.e., the compressed data can be decompressed to the original form with no loss of information) and lossy (i.e., the compression process loses some information, which becomes unrecoverable). Either way, the process must be reversible, providing a function that can fully or almost fully restore the original data.

The quality of a compression algorithm can be measured in terms on **compression ratio**, which is the ratio between the size of the original data and the size of the compressed data:

$$CR = \frac{|B|}{|C(B)|}$$

If $CR = r$, then the size of the compressed output is r times smaller than the input size. The compression ratio can be chosen to find the best tradeoff between compression

speed, amount of energy spent, loss of precision, etc. Ultimately, an important goal is to find a compression algorithm that allows direct computation over compressed data without the need for decompression.

6.1 Lossless Compression

6.1.1 Integer Encoders

We are given an integer $x > 0$, and want an algorithm, called **code**, that represents x in as few as possible bits. The output of the code is a bit-string called **codeword**, indicated as $C(x)$. A message $L = [x_1, \ldots, x_n]$ consisting of n integers can be coded as the concatenation of the codewords of each integer: $C(L) = [C(x_1), \ldots, C(x_n)]$. The codes we will see next are **static**, meaning they always assign the same codeword to the same integer regardless of the message to be coded.

Before applying any actual compression algorithm, the posting list can be preprocessed to reduce the value of the document IDs within them. Starting from the second element in the list, each docID is replaced by the difference between it and the previous one. This technique is called **d-gaps**.

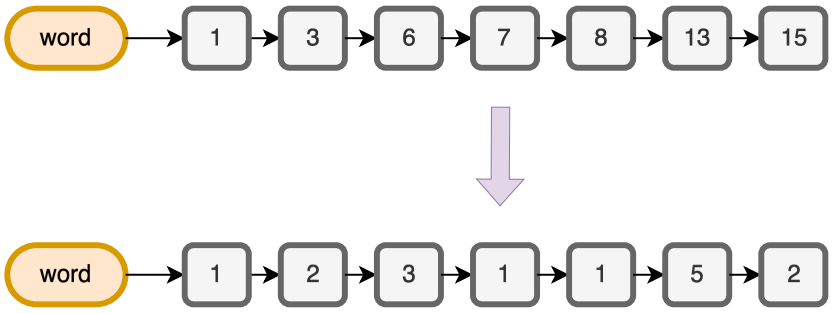

Figure 6.1: Example of d-gaps.

Binary Encoding Binary encoding converts each number to a binary string of fixed length. Given $bin(x)$ the function that returns the binary representation of x, the binary codewords are calculated as:

$$B(x) = bin(x - 1)$$

This way, each number can be represented using $|B(x)| = \lceil log_2(x) \rceil$ bits (the trailing zeros are removed). This major problem of this technique is that when the codewords are concatenated together to form the message, there is no way of telling where a codeword starts and where it ends: this leads to an ambiguous representation that does

not have a unique decoding. The source of ambiguity is caused by the fact that the code $B()$ is not **prefix-free**: a code C is said to be prefix-free if there are no x, y s.t. $C(x) \leq C(y)$ for which $C(x) = C(y)[1 : |C(x)|]$. In other words, this compression can make it so that some codewords are the prefix of other, longer codewords.

Unary Encoding Unary encoding produces sequences of bits set to 1, and uses 0 as the delimeter between each sequence. Each number is coded as:

$$U(x) = 1^{x-1}0$$

i.e., a sequence of $x - 1$ 1s and a 0. Therefore, $|U(x)| = x$. This method is good only for small integers, since the size of the compressed output grows linearly with that of the uncompressed input.

Elias Gamma and Delta Encoding The key idea on which the Elias Gamma encoding is based is to use the binary representation of a number, and prepend the unary encoding of its length. The encoding function is as follows:

$$\gamma(x) = U(|bin(x)|)bin(x)[2:]$$

The first bit is removed from the binary representation, since it is always equal to 1. For example, $\gamma(4) = 11\ 0\ 00$. The first two bits mean that the following word is 3 bits long; the next 0 bit is a delimiter; the last two bits are the binary encoding of number 4 (note that the first 1 bit was removed from 100).

Elias Delta encoding is built on top of Gamma. It still prepends the length of the binary representation, but it is further compressed using Elias Gamma. The code function is:

$$\delta(x) = \gamma(|bin(x)|)bin(x)[2:]$$

Using the same example as before, $\delta(4) = \gamma(3)00 = 10\ 1\ 00$. The first two bits mean that the binary representation of 3 is long 2 bits, the next bit is the binary representation of 3 (without the leading 1), and finally, the last two bits are the binary representation of 4 (again without the leading 1).

These methods can be extended one upon the other indefinitely; each further extension produces codewords that are shorter for bigger numbers, and longer for shorter numbers. Additionally, if a method chains m compressions, we also need to do m decompressions, thus increasing the time needed.

Variable Byte Encoding Variable Byte encoding produces codewords that are byte-aligned instead of bit-aligned; byte-alignment favors implementation simplicity and encoding/decoding speed at the cost of lower compression ratio.

This method uses the binary encoding of a number, and splits it into bytes; for each byte, the 7 least significant ones are used for the representation itself (**data bits**), while the most significant bit (**control bit**) is used to signal continuation (1) or end (0) of that representation. For example, $VB(4) = 0\ 0000011$. It only takes one byte since it's small. On the other hand, a large number can require multiple bytes: $VB(267) = 1\ 0000010\ 0\ 0001010$.

6.1.2 List Encoders

List encoders are used to compress an entire integer list instead of individual integers. The goal is to compress a list $L = [x_1, \ldots, x_n]$ of strictly increasing integers, where $x_1 > 0$ and $x_n \leq U$. U is called universe size. Some compressors operate directly on the list, while others first transform the list into a sequence of d-gaps.

As a way to analyze the space effectiveness of the compressors is to compare them to the combinatorial lower bound, meaning the minimum number of bits needed to represent a list of n strictly increasing integers drawn out of a universe of size $U \geq n$. There are $\binom{U}{n}$ ways of randomly choosing n distinct numbers out of U possibilities; therefore, we need at least $\lceil \log_2 \binom{U}{n} \rceil$ bits. By Stirling's approximation, we can estimate the lower bound as:

$$\left\lceil \log_2 \binom{U}{n} \right\rceil \approx n(\log_2 \left(\frac{U}{n}\right) + \log_2 e) \approx n \log_2 \left(\frac{U}{n}\right) + 1.44n\,.$$

In practice, however, many compressors actually take less space than this lower bound. This is because the lower bound assumes that the integers are truly randomly distributed, but in real applications, lists of document IDs tend to contain many clusters of close integers, and this closeness can be exploited. For example, in a inverted index built for a search engine, the word "Pisa" may contain a lot of IDs of documents relating to the University of Pisa, which are likely to be all close to each other.

Simple9 and Simple16 These two methods are based on the same principle: pack as many integers as possible in a memory word (32 or 64 bits long). Both operate on d-gaps. Simple9 adopts 32-bit words and has 9 "configurations", while Simple16 has 16 of them.

Simple 9 dedicates 4 bits to the **selector code** and the remaining 28 for the data. The selector indicates how many elements are packed in the segment using equally sized codewords. The possible configurations are described in the table below:

Selector	Integers	Bits per Integer
0000	28	1
0001	14	2
0010	9	3
0011	7	4
0100	5	5
0101	4	7
0110	3	9
0111	2	14
1000	1	28

Table 6.1: Simple9 selector codes.

For example, the list $L = 1^{33}\ 3^5\ 2^5\ 1^2\ 7^6$ is encoded with three 32-bit words as $1^{28}\ -\ 1^5\ 3^5\ 2^4\ -\ 2^1\ 1^2\ 7^6$. The very first block has selector 0000, since the only integer present is 1; the second block has selector 0001, since we need two bits to represent 3 and 2; the last block has selector 0010, since we need three bits to represent 7.

Patched Frame of Reference A big limitation of block-based strategy is their inefficiency when a block contains only a single large value, because it forces a large amount of bits to be used for the codeword: in the example above, had the last block contained the number 512 instead of 7, it would have forced the use of 9 bits for any other number in the same block (which, for small numbers such as 1 or 2, would have big a huge waste of space).

Patched Frame of Reference (PFor) tries to solve this issue by choosing a base b and a value $k > 0$ for the universe representation of the block, such that a large percentage (e.g., 90%) of the block's integers fall in the range $[b, b + 2^k - 1]$, i.e., they can be represented using k bits. Any integer that does not fit into k bits is treated as an **exception** and encoded in a separate array using some other compression algorithm.

Binary Interpolative Coding Suppose we have a list of integers L such that $l \leq L[1]$ and $L[n] \leq h$. Then, we can encode the element in the middle of the sequence, $L[m]$ (where $m = \lceil n/2 \rceil$), knowing it is in the range $[l, h]$. Instead of encoding this number as it is, we encode $L[m] - l$ using $\lceil \log_2(h - l) \rceil$ bits. For example, the list is $L = [2, 3, 5, 7, 9, 10, 13]$, then $l = 2$ and $h = 13$. The middle element is 7, and we can encode it as $7 - 2 = 5$. If we apply the same encoding to the two halves $L[1 : m - 1]$ and $L[m + 1 : n]$, we now have updated knowledge of who the upper and lower bounds are for each half. In the example, the two sublists are $L_1 = [2, 3, 5]$ ($l_1 = 2, h_1 = 6$) and

$L_2 = [9, 10, 13]$ ($l_2 = 8, h_2 = 13$).

The crucial property of this method is that whenever the length r of the interval is equal to $h - l$, then a run of r consecutive integers can be detected and no bits are necessary: for this reason, binary interpolative coding is particularly effective on highly clustered sequences.

Elias-Fano Encoding Let $S(n, U)$ be a sorted sequence of n integers drawn from a universe of size U. Each integer $S[i]$ is converted in its binary representation $bin(S[i])$, which takes $\lceil \log_2 U \rceil$ bits; this representation is split in two parts:

- The **low part** containing the low bits, which are the rightmost $l = \lceil \log_2(U/n) \rceil$ bits;
- The **high part** containing the high bits, which are the remaining $h = \lceil \log_2 U \rceil - l$ bits.

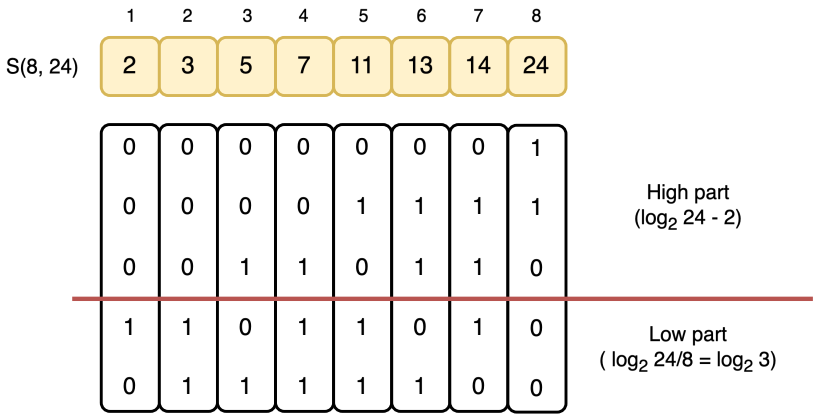

Figure 6.2: Example array split into high and low parts.

The encoding of $S(n, U)$ is given by the encoding of the high and low parts of all the integers. The low parts of the integers, $[l_1, \ldots, l_n]$, are written as they are in a bitvector L of $n \lceil \log_2(U/n) \rceil$ bits. The high parts $[h_1, \ldots, h_n]$ are also represented with a bitvector, of $n + 2^{\lfloor \log_2 n \rfloor} \leq 2n$ bits. We start from a 0-valued bitvector H, and set the bit in position $h_i + i$, $\forall i = 1, \ldots, n$. The end result is a vector that contains the cardinalities of the "buckets" in which the bits of the high part can be grouped into. This vector is composed of $\leq 2n$ bits because it contains a 1 for each number in the sequence, and no more than a 0 for each possible bucket (at most n).

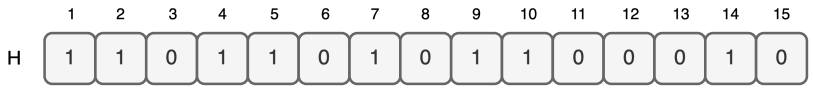

Figure 6.3: Vector H obtained from the example above. Note how the first bucket has two elements, the second also has two elements, the third has one element, and so on: these cardinalities are stored in sequence, separated by one or more 0s.

Finally, the Elias-Fano representation of S is the concatenation of H and L, and overall takes

$$EF(S(n,U)) \leq n \lceil \log_2(U/n) \rceil + 2n$$

bits.

Elias-Fano supports random access without decompression. To access an element, we want to implement the operation $\texttt{Access}(i)$ which returns $S[i]$. We assume we can efficiently use the $\texttt{Select}_0(i)$ and $\texttt{Select}_1(i)$ primitives, which return, respectively, the position of the i^{th} 0 and 1 in a bitvector. Using $\texttt{Select}_1(i)$, it is possible to implement $\texttt{Access}(i)$ in $O(1)$ time. To reconstruct a number, we need to re-link the two halves of its binary representation: the low part is easily retrieved, as it is the range of bits $L[(i-1)*l+1, i*l]$. The retrieval of the high bits requires a few steps; the bitvector H tells us how many integers share the same high part and fall in the same bucket, expressed in unary. To find exactly who are these bits for a given position in the original sequence, we simply need to know how many 0s are encountered before the i^{th} 1 in H; the position of this bit is retrieved using $\texttt{Select}_1(i)$. If we remove i from that position (essentially doing the inverse operation used to build the bitvector), we find the original value of the high bits. By concatenating them to the low bits found before, the whole number is reconstructed.

As an example, consider Figures 6.2 and 6.3. If we wanted to find the 4^{th} element of the sequence, we would first look at the low bits, stored as they are, reading from position 7 to 8 (finding bits 11). Then, we look at H, and find the position of the 4^{th} 1, which is 5. We subtract 4 from this position, obtaining 1, i.e., 001 in bit representation. Finally, we concatenate the two halves and find that the number in the 4^{th} position is $00111 = 7$.

Partitioned Elias-Fano Encoding As mentioned above in PFor, it's common for posting lists to have clusters of consecutive document IDs. Elias-Fano does not exploit the presence of these clusters, requiring the same amount of bits to encode a list regardless of its structure. Partitioned Elias-Fano encoding is a variant that first partitions each posting list into chunks of variable length, and then separately encodes each chunk

using Elias-Fano. The optimal partitioning that minimizes the size of compressed data can be found in linear time uning a dynamic programming algorithm.

Chapter 7

Learning to Rank

Bibliography