Lesson 4

Inverted Indexing for Text Retrieval

BIT-ALIGNED CODES

The advantage of byte-aligned and word-aligned codes is that they can be coded and decoded quickly. The downside, however, is that they must consume multiples of eight bits, even when fewer bits might suffice (the Simple-9 scheme gets around this by packing multiple integers into a 32-bit word, but even then, bits are often wasted). In bit-aligned codes, on the other hand, code words can occupy any number of bits, meaning that boundaries can fall anywhere. In practice, coding and decoding bit-aligned codes require processing.

BIT-ALIGNED CODES

Simple-9

Selector	Code length	Number of codes	Bits unused
0000	1	28	0
0001	2	14	0
0010	3	9	1
0011	4	7	0
0100	5	5	3
0101	7	4	0
0110	9	3	1
0111	14	2	0
1000	28	1	0

BIT-ALIGNED CODES

One additional challenge with bit-aligned codes is that we need a mechanism to delimit code words, i.e., tell where the last ends and the next begins, since there are no byte boundaries to guide us. To address this issue, most bit-aligned codes are so called

prefix codes (confusingly, they are also called prefix-free codes), in which no valid code word is a prefix of any other valid code word. For example, coding {0, 1, 2} with {0, 1, 01} is not a valid prefix code, since 0 (one of the codes) is a prefix of 01 (another code), and so we can't tell if 01 is two code words or one. On the other hand, {00, 01, 1} is a valid prefix code, such that a sequence of bits: 000110100101010 0 can be unambiguously segmented into: 00 01 1 01 00 1 01 01 00 and decoded without any additional delimiters.

BIT-ALIGNED CODES

One of the simplest prefix codes is the unary code. An integer $x > 0$ is coded as $x - 1$ one bits followed by a zero bit. Note that unary codes do not allow the representation of zero, which is fine since d-gaps and term frequencies should never be zero. As an example, 4 in unary code is 1110. With unary code we can code x in x bits, which although economical for small values, becomes inefficient for even moderately large values. Unary codes are rarely used by themselves, but form a component of other coding schemes.

BIT-ALIGNED CODES

Elias γ code is an efficient coding scheme that is widely used in practice. An integer $x > 0$ is broken into two components, $1 + \lfloor \log_2 x \rfloor$ (= n, the length), which is coded in unary code, and $x - 2^{\lfloor \log_2 x \rfloor}$ (= r, the remainder), which is in binary. The unary component n specifies the number of bits required to code x, and the binary component codes the remainder r in n - 1 bits. As an example, consider $x = 10$. $1 + \lfloor \log_2 10 \rfloor = 4$, which is 1110 in unary code. The binary component codes $x - 2^{\lfloor \log_2 10 \rfloor} = 10 - 8 = 2$ is coded in 4 - 1 = 3 bits, which is 010. Putting both together, we arrive at 1110010.

BIT-ALIGNED CODES

It is easy to unambiguously decode a bit stream of codes. First, we read a unary code n, which is a prefix code. This tells us that the binary portion is n - 1 bits, which we then read as r. Thus x is computed as $2^{n-1} + r$. For $x < 16$, codes occupy less than a full byte, which makes them more compact than variable-length integer codes. Since term frequencies for the most part are relatively small, γ codes make sense for them and can yield substantial space savings.

RETRIEVAL

It should be fairly obvious that serving the search needs of a large number of users, each of whom demand sub-second response times, is beyond the capabilities of any single machine. The only solution is to distribute retrieval across a large number of machines, which necessitates breaking up the index in some manner.

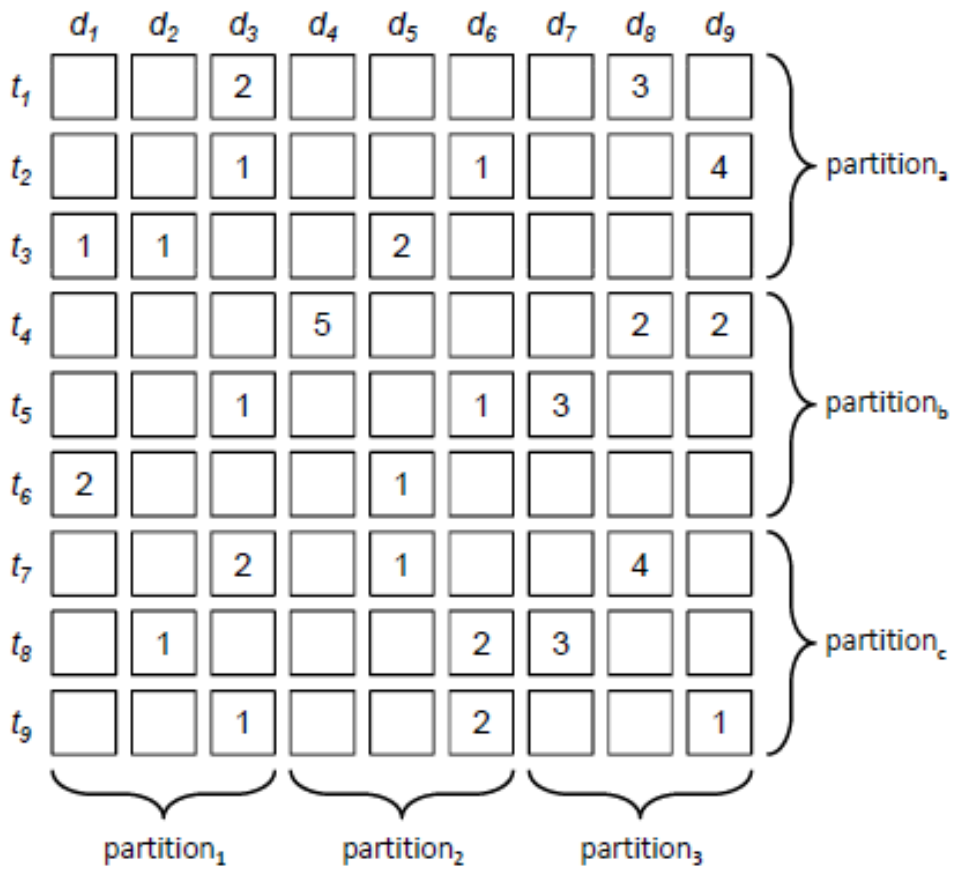
There are two main partitioning strategies for distributed retrieval: document partitioning and term partitioning. Under document partitioning, the entire collection is broken up into multiple smaller sub-collections, each of which is assigned to a server.

RETRIEVAL

The server holds the complete index for a subset of the entire collection. This corresponds to partitioning vertically in Figure 4.6. With term partitioning, on the other hand, each server is responsible for a subset of the terms for the entire collection. That is, a server holds the postings for all documents in the collection for a subset of terms. This corresponds to partitioning horizontally in Figure 4.6.

Figure 4.6: Term-document matrix (nine documents, nine terms). illustrating different partitioning strategies: partitioning vertically (1; 2; 3) corresponds to document partitioning, whereas partitioning horizontally (a; b; c) corresponds to term partitioning

RETRIEVAL


d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9
t_1 2 3
t_2 1 1 4
t_3 1 1 2
t_4 5 2 2
t_5 1 1 3
t_6 2 1
t_7 2 1 4
t_8 1 2 3
t_9 1 2 1
partition$_a$
partition$_b$
partition$_c$
partition$_1$ partition$_2$ partition$_3$

RETRIEVAL

Document and term partitioning require different retrieval strategies and represent different tradeoffs.

Retrieval under document partitioning involves a query broker, which forwards the user's query to all partition servers, merges partial results from each, and then returns the final results to the user. With this architecture, searching the entire collection requires that the query be processed by every partition server. However, since each partition operates independently and traverses postings in parallel, document partitioning typically yields shorter query latencies.

RETRIEVAL

In general, studies have shown that document partitioning is a better strategy overall [109], and this is the strategy adopted by Google [16]. Furthermore, it is known that Google maintains its indexes in memory (although this is certainly not the common case for search engines in general). One key advantage of document partitioning is that result quality degrades gracefully with machine failures. Partition servers that are offline will simply fail to deliver results for their subsets of the collection. With sufficient partitions, users might not even be aware that documents are missing.