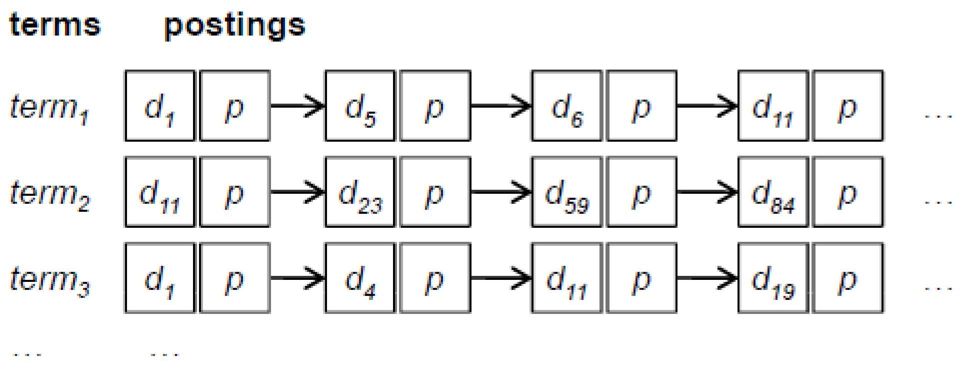
Lesson 4

Inverted Indexing for Text Retrieval

INVERTED INDEXES

An inverted index consists of postings lists, one associated with each term that appears in the collection.

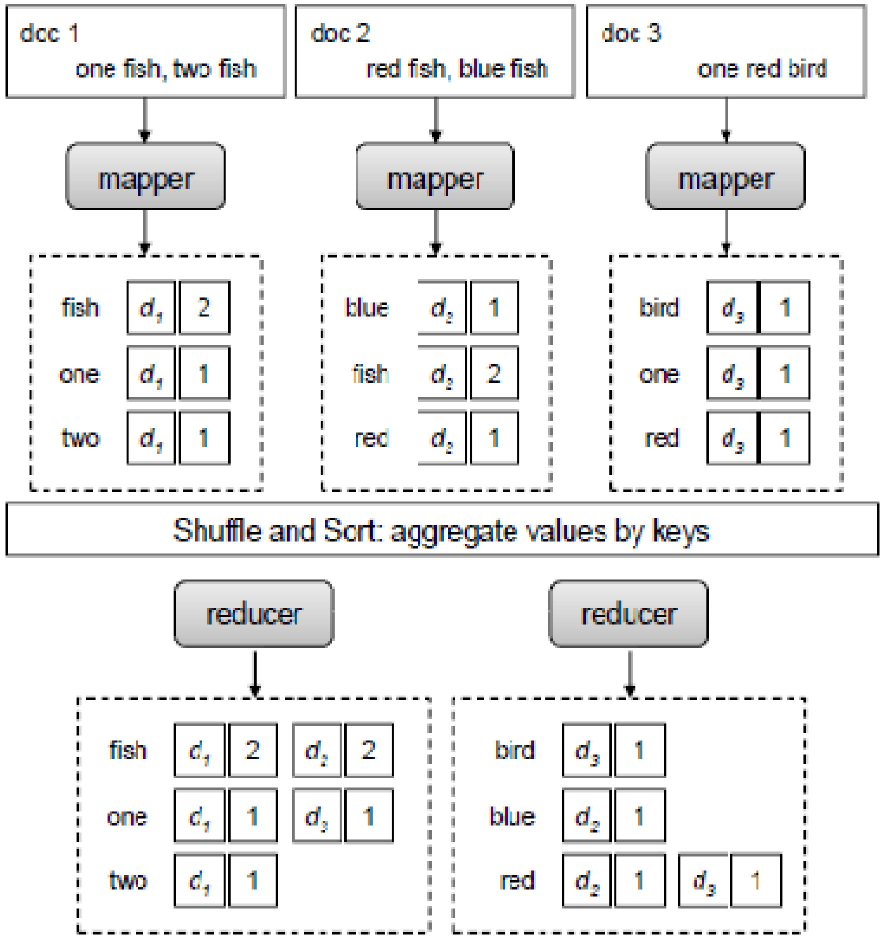

terms
postings
$term_1$
d_1 p → d_5 p → d_6 p → d_{11} p ...
$term_2$
d_{11} p → d_{23} p → d_{59} p → d_{84} p ...
$term_3$
d_1 p → d_4 p → d_{11} p → d_{19} p ...
... ...

A postings list is comprised of individual postings, each of which consists of a document id and a payload - information about occurrences of the term in the document. The simplest payload is “nothing”!

INVERTED INDEXING: BASELINE IMPLEMENTATION

```
class Reducer
    procedure Reduce(term t; postings [< n1, f1 >, < n2, f2>, …])
        P ← new List
        for all posting < a, f > in postings [< n1, f1 >, < n2, f2>, …] do
            Append(P, < a, f >)
        Sort(P)
        Emit(term t, postings P)
```

INVERTED INDEXING: BASELINE IMPLEMENTATION



INVERTED INDEXING: REVISED IMPLEMENTATION

The inverted indexing algorithm presented in the previous section serves as a reasonable baseline. However, there is a significant scalability bottleneck: the algorithm assumes that there is sufficient memory to hold all postings associated with the same term. Since the basic MapReduce execution framework makes no guarantees about the ordering of values associated with the same key, the reducer first buffers all postings (line 5 of the reducer pseudo-code in Figure 4.2) and then performs an in memory sort before writing the postings to disk. For efficient retrieval, postings need to be sorted by document id. However, as collections become larger, postings lists grow longer, and at some point in time, reducers will run out of memory.

INVERTED INDEXING: REVISED IMPLEMENTATION

There is a simple solution to this problem. Since the execution framework guarantees that keys arrive at each reducer in sorted order, one way to overcome the scalability bottleneck is to let the MapReduce runtime do the sorting for us. Instead of emitting key-value pairs of the following type:

(term t; posting < docid, f >)

We emit intermediate key-value pairs of the type instead:

(tuple <t, docid> tf f)

INVERTED INDEXING: REVISED IMPLEMENTATION

In other words, the key is a tuple containing the term and the document id, while the value is the term frequency. This is exactly the value-to-key conversion design pattern introduced in Section 3.4. With this modification, the programming model ensures that the postings arrive in the correct order. This, combined with the fact that reducers can hold state across multiple keys, allows postings lists to be created with minimal memory usage. As a detail, remember that we must define a custom partitioner to ensure that all tuples with the same term are shuffled to the same reducer.

INVERTED INDEXING: REVISED IMPLEMENTATION

The revised MapReduce inverted indexing algorithm is shown in Figure 4.4. The mapper remains unchanged for the most part, other than differences in the intermediate key-value pairs. The Reduce method is called for each key (i.e., < t, n>), and by design, there will only be one value associated with each key. For each key-value pair, a posting can be directly added to the postings list. Since the postings are guaranteed to arrive in sorted order by document id, they can be incrementally coded in compressed form - thus ensuring a small memory footprint.

INVERTED INDEXING: REVISED IMPLEMENTATION

Finally, when all postings associated with the same term have been processed (i.e., t ≠ tprev), the entire postings list is emitted. The final postings list must be written out in the Close method. As with the baseline algorithm, payloads can be easily changed: by simply replacing the intermediate value f (term frequency) with whatever else is desired (e.g., term positional information).

INVERTED INDEXING: REVISED IMPLEMENTATION

There is one more detail we must address when building inverted indexes. Since almost all retrieval models take into account document length when computing query - document scores, this information must also be extracted. Although it is straightforward

to express this computation as another MapReduce job, this task can actually be folded into the inverted indexing process. When processing the terms in each document, the document length is known, and can be written out as “side data" directly to HDFS.

INVERTED INDEXING: REVISED IMPLEMENTATION (CONT'D)

We can take advantage of the ability for a mapper to hold state across the processing of multiple documents in the following manner: an in-memory associative array is created to store document lengths, which is populated as each document is processed. When the mapper finishes processing input records, document lengths are written out to HDFS (i.e., in the Close method). This approach is essentially a variant of the in-mapper combining pattern. Document length data ends up in m different files, where m is the number of mappers; these files are then consolidated into a more compact representation. Alternatively, document length information can be emitted in special key-value pairs by the mapper. One must then write a custom partitioner so that these special key-value pairs are shuffled to a single reducer, which will be responsible for writing out the length data separate from the postings lists.

INVERTED INDEXING: REVISED IMPLEMENTATION

```
class Mapper
    method Map(docid n; doc d)
    H ← new AssociativeArray
    for all term t in doc d do
        H{t} ← H{t} + 1
    for all term t in H do
        Emit(tuple < t, n>, tf H{t})
```

INVERTED INDEXING: REVISED IMPLEMENTATION

```
class Reducer
    method Initialize
        tprev ← Ø, P ← new PostingsList
    method Reduce(tuple < t, n >,  tf [f])
        if (t ≠ tprev && tprev ≠ Ø) then
            Emit(term tprev, postings P),    P.Reset()
        P.Add(< n, f >), tprev ←  t
    method Close
        Emit(term tprev; postings P)
```

INDEX COMPRESSION

We return to the question of how postings are actually compressed and stored on disk. This chapter devotes a substantial amount of space to this topic because index compression is one of the main differences between a “toy" indexer and one that works on real-world collections. Otherwise, MapReduce inverted indexing algorithms are pretty straightforward.

INDEX COMPRESSION

Let us consider the canonical case where each posting consists of a document id and the term frequency. A naive implementation might represent the first as a 32-bit integer and the second as a 16-bit integer.

[(5, 2), (7, 3), (12, 1), (49, 1), (51, 2), …]

Note that all brackets, parentheses, and commas are only included to enhance readability; in reality the postings would be represented as a long stream of integers.

INDEX COMPRESSION

This naive implementation would require six bytes per posting. The entire inverted index would be about as large as the collection itself.

The first idea is to encode differences between document ids. Since the postings are sorted by document ids, the differences (called d-gaps) are positive integers. Thus the postings list,

[(5, 2), (7, 3), (12, 1), (49, 1), (51, 2), …], represented with d-gaps is

[(5, 2), (2, 3), (5, 1), (37, 1), (2, 2), …]

INDEX COMPRESSION

We must actually encode the first document id. We haven't lost any information, since the original document ids can be easily reconstructed from the d-gaps. However, it's not obvious that we've reduced the space requirements either, since the largest possible d-gap is one less than the number of documents in the collection. This is where the second idea comes in, which is to represent the d-gaps in a way such that it takes less space for smaller numbers. We want to apply the same techniques to the term frequencies, since for the most part they are also small values. But to understand, take a slight detour into compression techniques for coding integers.

INDEX COMPRESSION

Compression, in general, can be characterized as either lossless or lossy: it's fairly obvious that loseless compression is required in this context. To start, it is important to understand that all compression techniques represent a time/space tradeoff. That is, we reduce the amount of space on disk necessary to store data, but at the cost of extra processor cycles that must be spent coding and decoding data. Therefore, it is possible that compression reduces size but also slows processing. However, if the two factors are properly balanced (i.e., decoding speed can keep up with disk bandwidth), we can achieve the best of both worlds: smaller and faster.

BYTE-ALIGNED AND WORD-ALIGNED CODES

In most programming languages, an integer is encoded in four bytes and holds a value between 0 and $2^{32} - 1$, inclusive. We limit our discussion to unsigned integers, since d-gaps are always positive. This means that 1 and 4,294,967,295 both occupy four bytes. Obviously, encoding d-gaps this way doesn't yield any reductions in size.

A simple approach to compression is to only use as many bytes as is necessary to represent the integer. This is known as **variable-length integer coding** (**varInt** for short) and accomplished by using the high order bit of every byte as the continuation bit, which is set to one in the last (lowest) byte and zero elsewhere.

BYTE-ALIGNED AND WORD-ALIGNED CODES

As a result, we have 7 bits per byte for coding the value, which means that $0 \le n < 2^7$ can be expressed with 1 byte, $2^7 \le n < 2^{14}$ with 2 bytes, $2^{14} \le n < 2^{21}$ with 3, and $2^{21} \le n < 2^{28}$ with 4 bytes. This scheme can be extended to code arbitrarily-large integers (i.e., beyond 4 bytes). As a concrete example, the two numbers:

127, 128

would be coded as such:

1 1111111, 0 0000001 1 0000000

BYTE-ALIGNED AND WORD-ALIGNED CODES

The above code contains two code words, the first consisting of 1 byte, and the second consisting of 2 bytes. Of course, the comma and the spaces are there only for readability. Variable-length integers are byte-aligned because the code words always fall along byte boundaries. As a result, there is never any ambiguity about where one code word ends and the next begins. However, the downside of varInt coding is that decoding involves lots of bit operations. Furthermore, the continuation bit sometimes results in frequent branch mispredicts (depending on the actual distribution of d-gaps), which slows down processing.

BYTE-ALIGNED AND WORD-ALIGNED CODES

In most architectures, accessing entire machine words is more efficient than fetching all its bytes separately. Therefore, it makes sense to store postings in increments of 16-bit, 32-bit, or 64-bit machine words. Anh and Moat [8] presented several word aligned coding methods, one of which is called Simple-9, based on 32-bit words. In this coding scheme, four bits in each 32-bit word are reserved as a selector. The remaining 28 bits are used to code actual integer values. Now, there are a variety of ways these 28 bits can be divided to code one or more integers: 28 bits can be used to code one 28-bit integer, two 14-bit integers, three 9-bit integers (with one bit unused), etc., all the way up to twenty-eight 1-bit integers.

BYTE-ALIGNED AND WORD-ALIGNED CODES

In fact, there are nine different ways the 28 bits can be divided into equal parts (hence the name of the technique), some with leftover unused bits. This is stored in the selector bits. Therefore, decoding involves reading a 32-bit word, examining the selector to see how the remaining 28 bits are packed, and then appropriately decoding each integer. Coding works in the opposite way: the algorithm scans ahead to see how many integers can be squeezed into 28 bits, packs those integers, and sets the selector bits appropriately.