Web Services and Web Data

XJCO3011

Session 13 – Query Processing

Query Processing

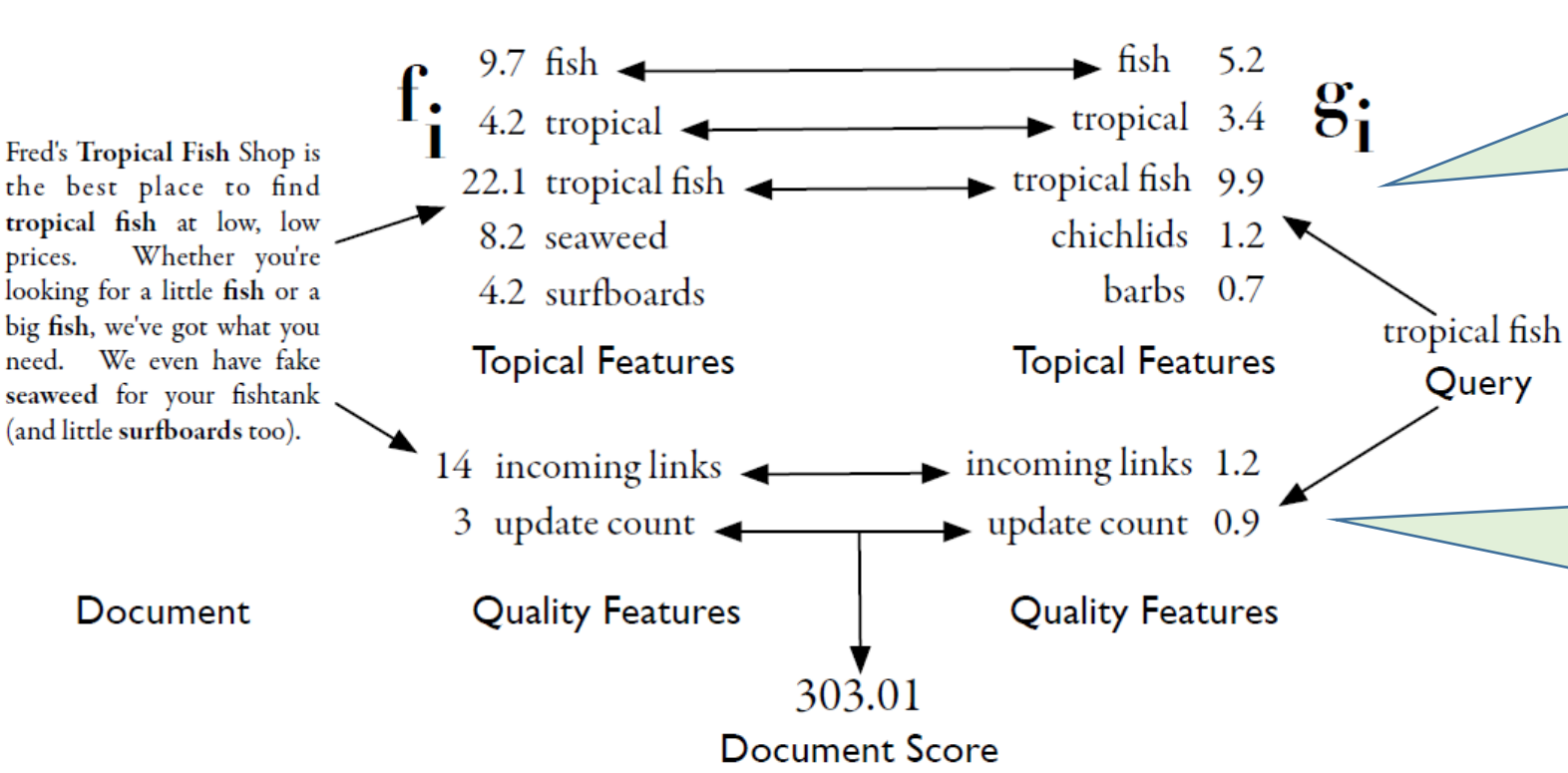
- Once an index is built, we need to process the data in it to produce query results.
- Even with simple algorithms, processing queries using an index is much faster than without one.
- However, clever algorithms can boost query processing speed by ten to a hundred times over the simplest versions.
- We will explore the simplest two query processing techniques first, called document-at-a-time and term-at-a-time, and then move on to faster and more flexible variants.

The General Ranking Model

- We assume that the ranking function R takes the following form:

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

- Here, f_i is some feature function that extracts a number from the document text, and g_i is a similar feature function that extracts a value from the query

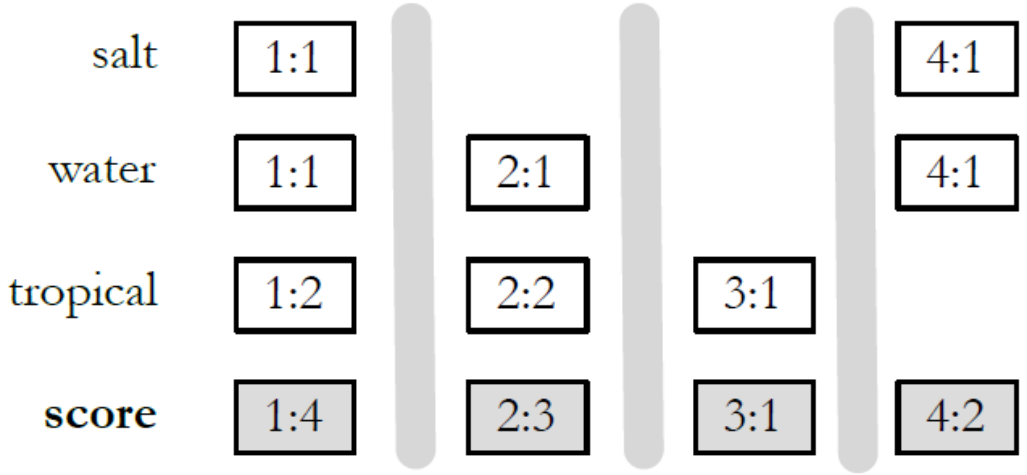


The feature function $g_{tropical}$(Q) evaluates to a large value because "tropical" is in the query. However, g_{barbs}(Q) also has a small non-zero value because it is related to other terms in the query.

How important document updates are to relevance for this query. For instance, if the query was "today's weather in London", we would prefer documents that are updated frequently

Document-at-a-time Evaluation

- Document-at-a-time retrieval is the simplest way to perform retrieval with an inverted file.
- The figure below shows a document-at-a-time retrieval for the query "salt water tropical".
- The inverted lists are shown horizontally, and the postings have been aligned so that each column represents a different document
- The inverted lists hold word counts, and the score is just the sum of the word counts in each document.
- The vertical grey lines indicate the different steps of retrieval
- In the first step, all the counts for the first document are added to produce the score for that document.
- Once the scoring for the first document has completed, the second document is scored, then the third, and then the fourth.



Document-at-a-time Pseudocode

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Where Q: the query; I: the index; f and g: the feature functions; and k: the number of documents to retrieve

For each word w_i in the query, an inverted list is fetched from the index. These inverted lists are assumed to be sorted in order by document number. All of the fetched inverted lists are stored in an array, L.

Iterate for each doc. in the Index

For each document, all of the inverted lists are checked. If the document appears in one of the inverted lists, the feature function f_i is evaluated, and the document score s_D is computed by adding up the weighted function values

the document score is added to the priority queue R.

Document-at-a-time Algorithm Features

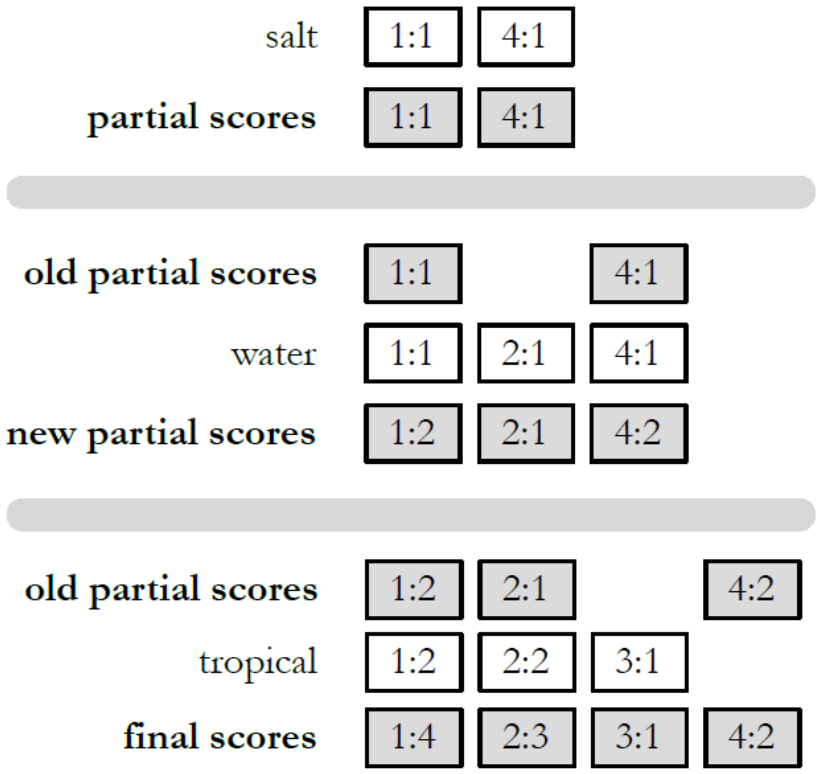
```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)        ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

The primary benefit of this method is its economic use of memory. The only major use of memory comes from the priority queue, which only needs to store k entries at a time. However, in a realistic implementation, large portions of the inverted lists would also be buffered in memory during evaluation.

Looping over all documents in the collection is unnecessary; we can change the algorithm to score only documents that appear in at least one of the inverted lists.

the priority queue R only needs to hold the top k results at any one time. If the priority queue ever contains more than k results, the lowest-scoring documents can be removed until only k remain, in order to save memory.

Term-at-a-time Evaluation



- The figure shows term-at-a-time retrieval, using the same query, scoring function, and inverted list data as in the document-at-a-time example.
- Notice that the computed scores are exactly the same in both figures, although the structure of each figure is different.
- As before, the grey lines indicate the boundaries between each step. In the first step, the inverted list for “salt” is decoded, and partial scores are stored in accumulators.
- These scores are called partial scores because they are only a part of the final document score
- In the second step, partial scores from the accumulators are combined with data from the inverted list for “water” to produce a new set of partial scores.
- After the data from the list for “tropical” is added in the third step, the scoring process is complete.
- In practice accumulators are stored in a hash table. The information for each document is updated as the inverted list data is processed

term-at-a-time retrieval algorithm

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q) f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

For each word w_i in the query, an inverted list is fetched from the index. These inverted lists are assumed to be sorted in order by document number. All of the fetched inverted lists are stored in an array, L.

the outer loop is over each list

reads each posting of the list, computing the feature functions fi and gi and adding its weighted contribution to the accumulator Ad.

After the main loop completes, the accumulators are scanned and added to a priority queue, which determines the top k results to be returned.

term-at-a-time retrieval algorithm features

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q) f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

The primary disadvantage of the term-at-a-time algorithm is the memory usage required by the accumulator table A. Remember that the document-at-a-time strategy requires only the small priority queue R, which holds a limited number of results.

However, the term-at-a-time algorithm makes up for this because of its more efficient disk access. Since it reads each inverted list from start to finish, it requires minimal disk seeking, and it needs very little list buffering to achieve high speeds. In contrast, the document-at-a-time algorithm switches between lists and requires large list buffers to help reduce the cost of seeking.

In practice, neither the document-at-a-time nor term-at-a-time algorithms are used without additional optimizations. These optimizations dramatically improve the running speed of the algorithms, and can have a large effect on the memory footprint.

Optimization Techniques

- There are two main classes of optimizations for query processing.
- The first is to read less data from the index, and the second is to process fewer documents.
- The two are related, since it would be hard to score the same number of documents while reading less data.
- When using feature functions that are particularly complex, focusing on scoring fewer documents should be the main concern.
- For simple feature functions, the best speed comes from ignoring as much of the inverted list data as possible.

Conjunctive Processing

- One of the simplest kind of query optimization is conjunctive processing.
- By conjunctive processing, we just mean that every document returned to the user needs to contain all of the query terms.
- Conjunctive processing is the default mode for many web search engines, in part because of speed and in part because users have come to expect it.
- With short queries, conjunctive processing can actually improve efficiency.
- In contrast, search engines that use longer queries, such as entire paragraphs, will not be good candidates for conjunctive processing.
- Conjunctive processing works best when one of the query terms is rare, as in the query “fish locomotion”. The word “fish” occurs about 100 times as often as the word “locomotion”.
- Since we are only interested in documents that contain both words, the system can skip over most of the inverted list for “fish” in order to find only the postings in documents that also contain the word “locomotion”.
- Conjunctive processing can be employed with both term-at-a-time and document-at-a-time systems.

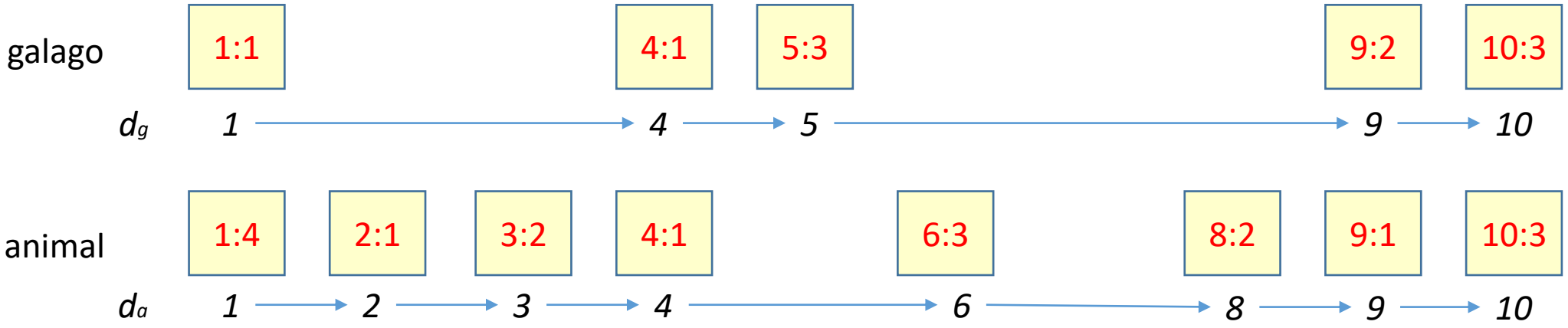
Processing Conjunctive Queries

- For many queries, we don't need all of the information stored in a particular inverted list. Instead, it would be more efficient to read just the small portion of the data that is relevant to the query
- Consider the Boolean query "galago AND animal". The word "animal" occurs in about 300 million documents on the Web versus approximately 1 million for "galago."
- If we assume that the inverted lists for "galago" and "animal" are in document order, there is a very simple algorithm for processing this query (next slide)

Galagos

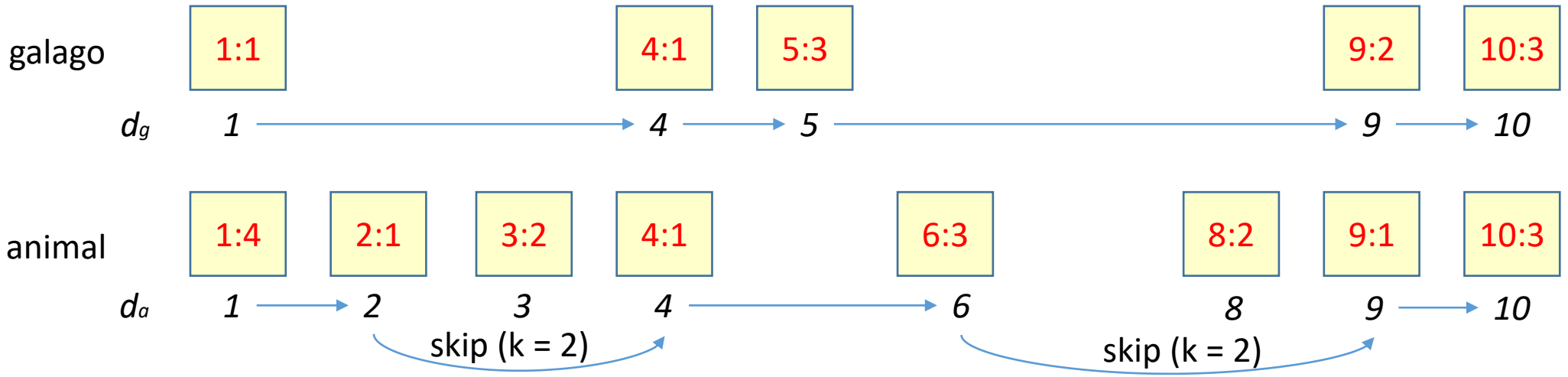
Processing Conjunctive Queries, Simple Algorithm

- Let d_g be the first document number in the inverted list for "galago."
- Let d_a be the first document number in the inverted list for "animal."
- While there are still documents in the lists for "galago" and "animal," loop:
  - If *da* = *dg*, the document *da* contains both "galago" and "animal". Move both *dg* and *da* to the next documents in the inverted lists for "galago" and "animal," respectively.
  - If $d_a < d_g$, set d_a to the next document number in the "animal" list.
  - If $d_g < d_a$, set d_g to the next document number in the "galago" list.



List Skipping

- Unfortunately, the previous algorithm is very expensive. It processes almost all documents in both inverted lists, so we expect the computer to process this loop about 300 million times.
- Over 99% of the processing time will be spent processing the 299 million documents that contain "animal" but do not contain "galago."
- We can change this algorithm slightly by skipping forward in the "animal" list.
- Every time we find that da < dg, we skip ahead k documents in the "animal" list to a new document, sa. If sa < dg, we skip ahead by another k documents. We do this until sa ≥ dg. At this point, we have narrowed our search down to a range of k documents that might contain dg, which we can search linearly. (Chap. 5.4.7)



Skip Pointers

- List skipping can be achieved with skip pointers
- Skipping, however, does not improve the asymptotic running time of reading an inverted list.
- Suppose we have an inverted list that is n bytes long, but we add skip pointers after each c bytes, and the pointers are k bytes long.
- Reading the entire list requires reading(n) bytes, but jumping through the list using the skip pointers requires Θ (kn/c) time, which is equivalent to Θ (n).
- Even though there is no asymptotic gain in runtime, the factor of c can be huge.
- For typical values of c = 100 and k = 4, skipping through a list results in reading just 2.5% of the total data.

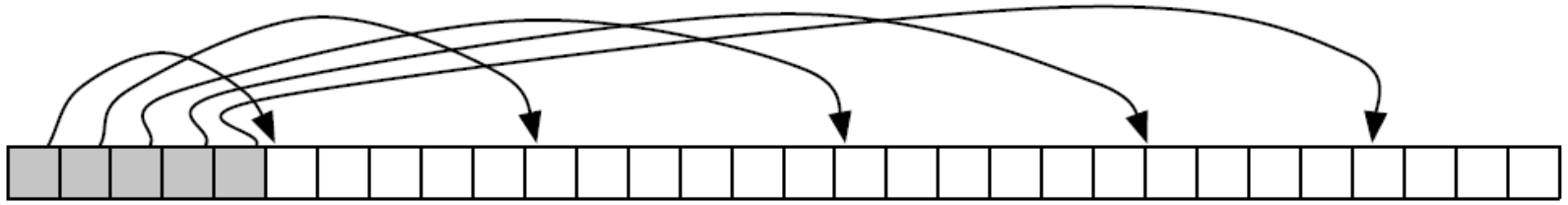


Fig. 5.19. Skip pointers in an inverted list. The gray boxes show skip pointers, which point into the white boxes, which are inverted list postings.

List Skipping Optimization

- Notice that in kn/c, as c gets bigger, the amount of data you need to read to skip through the list drops. So, why not make c as big as possible?
- The problem is that if c gets too large, the average performance drops. Let's look at this problem in more detail.
- Suppose you want to find p particular postings in an inverted list, and the list is n bytes long, with k-byte skip pointers located at c-byte intervals. Therefore, there are n/c total intervals in the list. To find those p postings, we need to read kn/c bytes in skip pointers, but we also need to read data in p intervals. On average, we assume that the postings we want are about halfway between two skip pointers, so we read an additional pc/2 bytes to find those postings. The total number of bytes read is then:

$$\frac{kn}{c} + \frac{pc}{2}$$

- you can see that while a larger value of c makes the first term smaller, it also makes the second term bigger.
- Therefore, picking the perfect value for c depends on the value of p, and we don't know what p is until a query is executed.
- However, it is possible to use previous queries to simulate skipping behaviour and to get a good estimate for c.

term-at-a-time algorithm for conjunctive processing

```
1:  procedure TermAtATimeRetrieval(Q, I, f, g, k)
2:      A ← Map()
3:      L ← Array()
4:      R ← PriorityQueue(k)
5:      for all terms w_i in Q do
6:          l_i ← InvertedList(w_i, I)
7:          L.add( l_i )
8:      end for
9:      for all lists l_i ∈ L do
10:         d_0 ← −1
11:         while l_i is not finished do
12:             if i = 0 then
13:                 d ← l_i.getCurrentDocument()
14:                 A_d ← A_d + g_i(Q) f(l_i)
15:                 l_i.moveToNextDocument()
16:             else
17:                 d ← l_i.getCurrentDocument()
18:                 d′ ← A.getNextAccumulator(d)
19:                 A.removeAccumulatorsBetween(d_0,d′)
20:                 if d = d′ then
21:                     A_d ← A_d + g_i(Q) f(l_i)
22:                     l_i.moveToNextDocument()
23:                 else
24:                     l_i skipForwardToDocument(d′)
25:                 end if
26:                 d_0 ← d′
27:             end if
28:         end while
29:     end for
30:     for all accumulators A_d in A do
31:         s_d ← A_d                ▷ Accumulator contains the document score
32:         R.add( s_d, d )
33:     end for
34:     return the top k results from R
35: end procedure
```

When processing the first term, (i = 0), processing

proceeds normally. However, for the remaining terms, (i > 0), the algorithm processes postings starting at line 17

It checks the accumulator table for the next

document that contains all of the previous query terms, and instructs list l_i to skip forward to that document if there is a posting for it. If there is a posting, the accumulator is updated. If the posting does not exist, the accumulator is deleted

That's it Folks

Further Reading

Chapter 5: Search Engines Information Retrieval in Practice by W. Bruce Croft, Donald Metzler, and Trevor Strohman