Algorithms in the Real World
Indexing and Search Engines II

Scribe notes by Lu Luo and Yangdong Deng
Based on a lecture by Guy Blelloch

December 13, 2001

- AND, OR, and NOT of terms (Fast Union, Intersection, Difference)
- Vector Space Models
- Signature Files

1 Union, Intersection and Difference Operations on Lists

In this section, we will discuss how to make efficient set operations (union, intersection and difference) on queries. With the inverted-lists structure used in search engines, the list for each term can be thought of as a set, since the order how documents are stored in the list does not necessarily affect the searching result. The query involving expressions of **AND**s and **OR**s on terms can be translated to set operations intersections and unions on the corresponding sets of documents (set difference can be used for **ANDNOT** queries).

Given two sets, these set operations can be implemented in time that is sublinear in the sum of the input set sizes. Extending the results shown by Brown and Tarjan that merging two lists X and Y of size n and m correspondingly, with m less than or equal to n, takes $O(m\log(\frac{n+m}{m}))$ comparisons, set union, intersection and difference can be implemented in the same bounds. This sublinear time is particularly important when one list is much longer than the other, where the time bound is $O(\log\ n)$. For general inputs, the $O(mlog(\frac{n+m}{m}))$ bound is a lower bound for the list merging problem.

To implement this effectively, additional constraints should be added to the algorithm mentioned above. First, the set operations on lists should be fully persistent, i.e. the original term lists should not be ruined, since the Brown and Tarjan's algorithm basically inserts the elements from one list into the other . The result set will share data with the input without modifying the input. The second issue is space, what if we want to include compression?

1.1 Merging Lists with Split and Join

The merge process can be thought of as cutting the two lists to the minimum number of consecutive subsequences or blocks necessary to achieve the merge, and then interleaving the two sets of blocks to form the sorted output. For the convenience of discussing, we assume that the documents in lists are sorted as discussions followed.

Definition 1 Let S be a set, given a value v in this set, the SPLIT operation on set S is:

$Split(S,\, v)$ = split S into two sets $S_<$ and $S_>$ with keys less than or equal to v, and return a flag saying whether v was present in the set.

Definition 2 Let S_1 and S_2 are sorted lists, assume they do not overlap. The JOIN operation on S_1 and S_2 is:

$Join(S_1,\, S_2)$ = join S_1 and S_2 into a new set, assuming $s_1 < s_2$ for all $s_1 \in S_1$ and $s_2 \in S_2$. Obviously one of the value in s_1, s_2 pair could be empty.

A balanced tree data structure can be used for representing a sorted sequence. To perform the SPLIT operation, the tree is split from v and work up the tree, duplicating every ancestor until either the leftmost or the rightmost branch of S is reached. One of each of these pairs of nodes, together with all of the items to the left of the duplicated node, eventually forms the tree $S_<$. The other node of the pair and items to the right of it forms $S_>$. At each level the children of each duplicated node are partitioned between the original node and the duplicate node according to its key value.

The time bound for SPLIT is:

$$O(\log(\min(|S_<|, |S_>|)))$$

which says that we can do it better and get the $O(\log(\text{smaller of the two subtrees}))$ bound for both SPLIT and or JOIN.

1.2 Implementing Union and Intersection Operations

Based on the SPLIT and JOIN operations introduced in last section, the UNION operation on two list S_1 and S_2 can be implemented as:

function $Union(S_1, S_2)$
  if $IsEmpty(S_1)$ **return** S_2
  else
    $(S_{2<}, S_{2>}, \text{flag}) = Split(S_2, \text{First}(S_1))$
  return $Join(S_{2<}, Union(S_{2>}, S_1))$

The function returns the ordered result of OR operation on two sets. If S_1 is empty, the function returns S_2; otherwise, it splits set S_2, using the first element in S_1 as the value v. If the first element in S_1 is in S_2, the variable $flag$ is set. In this way all elements in the new subset $S_{2<}$ are less than or equal to the first element of S_1, and all elements in the new subset $S_{2>}$ are greater than the first element of S_1. $S_{2<}$ becomes the new leftmost branch in the balanced tree. The function then work on the larger part of S_2 and S_1 to put the first element of S_2 into its proper position. Recursively, the algorithm ends when all elements in two sets are properly sorted and merged.

Likewise, the intersection is implemented as:

function $Intersect(S_1, S_2)$
  if $IsEmpty(S_1)$ **return** 0
  else
    $(S_{2<}, S_{2>}, \text{flag}) = Split(S_2, \text{First}(S_1))$
  if $flag$ **return** $Join(First(S_1), Intersect(S_{2>}, S_1))$
  else
    return $Intersect(S_{2>}, S_1)$

Intersect function returns the ordered result of AND operation on two sets. If one of the sets is empty then the intersection is empty; otherwise, it works just like the union algorithm. The difference is each time before recursion it put the element that exists in both sets into the result.

1.3 Runtime Analysis of Union Operation

As stated above, the set S resulted from $Union$ operation on lists S_1 and S_2 is an interleaving of blocks B_L, as illustrated as in Figure 1.

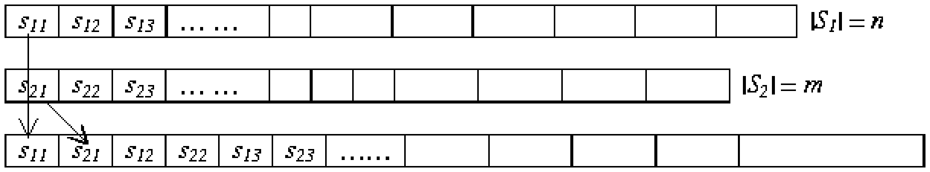


Figure X1. Illustration of blocks to be merged in union operation

The runtime for *Union* is:

$$T_{union} = O(\underbrace{\sum_i \log|B_L|}_{Splits} + \underbrace{\sum_i \log|B_L|}_{Joins})$$

Since logarithm is convex, this is maximized when blocks are as close as possible to equal size, and divided as finely as possible. We then have:

$$T_{union} = O(\sum_{i=1}^{m} \log\lceil \frac{n}{m} + 1\rceil) = O(m \log \frac{n+m}{m})$$

which is mentioned in the first section.

2 Treaps

There have been studies on fast and functional (i.e. persistence) algorithms for SPLIT and JOIN. The most contributive researches among them are:

- Cole 1995: Dynamic finger conjecture for Splay Trees is given. It gives amortized time bound but the proof is very involved.
- Kaplan and Tarjan 1997: The structure based on balanced trees is given and pointed out to be suitable to implement the list-merging algorithm with an optimal time bound. However the data structure they use is somewhat complex.

  Here $m = min(|S_1|, |S_2|)$, $n = max(|S_1|, |S_2|)$
- Today: Blandford and Blelloch develop very simple functional data structures for representing sets based on Treaps.

2.1 Definitions

A *Treap* is a balanced tree in which the key of each node has an associated random priority (specific relationship is not necessary between the key and priority). Nodes in a Treap are ordered according to the keys in the same way as is in a binary search tree. Recall that binary search tree has the following property:

Let x be a node in a binary search tree. If y is a node in the left subtree of x and z in the right, then $key(y) \leq key(x) \leq key(z)$.

If viewed by looking at the priorities assigned to nodes, a Treap is also a max -heap (and this is how the name "tre(e-he)ap" come into being). In a max-heap the priority of one node is greater than that of its children. When both the keys and priorities of a given set of nodes are unique, there is a unique Treap made up of these nodes. In other words, the structure of a Treap is independent of the order in which those keys are inserted. An example of a set of key- priority pairs and its corresponding Treap is shown in Figure 2.

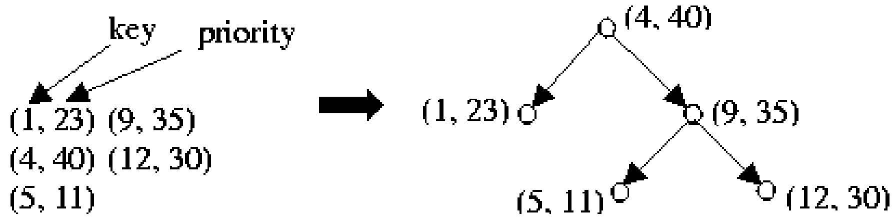


Figure 2. A Treap example

If both keys and priorities of a given set of nodes are unique, there exists a unique treap that contains these nodes. In other words, a treap is independent of the order in which its keys are inserted. This can be proved by induction. Since it is a heap, a treap's root node x must have the highest priority. Because it's a binary search tree as well, any node y with $key(y) < key(x)$ must be in the left subtree of x, and any node z with $key(x) < key(z)$ must be in the right subtree of x. Finally, since all subtrees are also treaps, by induction, their structures are fully determined.

Two operations, SPLIT and JOIN, are used to manipulate Treaps. SPLIT takes a T reap T and a key v, and returns a triple (T_1, v', T_2), such that T_1 and $T2$ are sub-Treaps of T, with all keys in T_1 less than v, all keys in T_2 larger than v. v' is a duplicate of v, if v is in T. The second operation JOIN, takes $T1$ a nd T_2 (let T_1 and T_2 be two Treaps such that all keys in T_1 are less than any k ey in T_2), and returns a single Treap T containing the keys in $T_1 U T_2$.

2.2 The Left Spinal Treap

In spinal Treap each edge is assigned a direction, and is called a pointer. A left spinal Treap (LTreap) is a Treap in which every pointer along the left spine is flipped, i.e. each node on the left spine has a pointer to its right child and a pointer to its parents. An example of LTreap is shown in Figure 3.

All searches in the LTreap structure start from the leftmost node, and we maintain a handle into it. By starting searching from leaves instead of the root takes much shorter time to run the SPLIT and JOIN operations.

Theorem: The time complexity of SPLIT operation (JOIN operation expects the sam e time complexity) on an LTreap is equal to the number of edges along the path from the start node to the split location. This path has length $log\ l$, where l is the expected number of nodes between start node and the split location in sorted order.

Proof:

Let $P(x)$ = length of path to node x from start

$$A_{ij} = \begin{cases} 1 & \text{if } x_i \text{ is an ancestor of } x_j; \\ 0 & \text{otherwise.} \end{cases}$$

$$C_{clm} = \begin{cases} 1 & \text{if } x_i \text{ is the common ancestor of } x_j \text{ and } x_m; \\ 0 & \text{otherwise.} \end{cases}$$

$$a_{ij} = E_x[A_{ij}],\ c_{clm} = E_x[C_{clm}],\ p(x) = E_x[P(x)]$$
$$P(x_l) = \sum_{i=1}^{l} A_{i1} + sum_{i=1}^{l}(A_{il} - C_{i1l})$$
$$E_x[P(x)] = p(x) = \sum_{j=1}^{l} a_{j1} + \sum_{j=1}^{l}(a_{il} + C_{i1l})$$

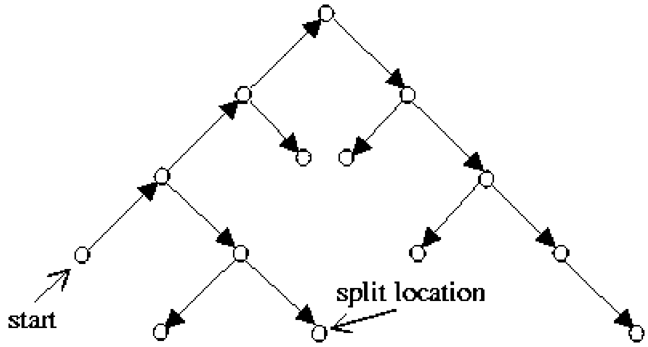


Figure 3. A left spinal Treap

Lemma: $a_{ij} = \frac{1}{|i-j|+1}$
Proof:

1. i is an ancestor of j, iff i has greater priority than all elements between i and j inclusively.
2. there are $|i-j|+1$ such elements each with equal probability of having the highest priority.

$$\sum_{j=1}^{l} a_{cl} = \underbrace{\sum_{j=1}^{l} \frac{1}{j} < (1 + log\ l)}_{\text{thormonica numbers } H_l}$$

We can similarly show that:

$$\sum_{j=1}^{n} a_{cl} + c_{c1l} = O(log\ l)$$

We leave this proof to exercise.

Therefore the expected path length and runtime for JOIN or SPLIT is $O(log\ n)$.

Similar techniques can be used for many properties of Treaps:
expected size of subtree $= O(log\ n)$
expected depth $= O(log\ n)$
...

...

Example: depth

$$D(x_j) = \sum_{i=1}^{n} A_{ij}$$

$$E_x[D(x_j)] = \sum_{i=1}^{n} a_{ij} = \sum_{i=1}^{n} \frac{1}{|i-j|+1} = H_j + H_{n-j} - 1 = O(log\ n)$$

This means expected search time for a Treap is $O(log\ n)$.

2.3 What about Space

For large database, huge number of documents can be stored in treap structure. Hence, the size of a treap can be potentially large. To reduce storage space requirement, an idea is to compress a part of the treap. Whether to compress a node in a treap is determined by its priority. If priorities of a treap are in range [0, 1], then any node with priority less than $1 - \sigma$ is stored compressed. σ should be set to represent the fraction of uncompressed nodes.

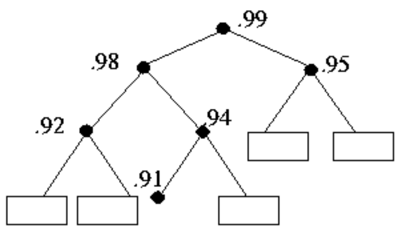
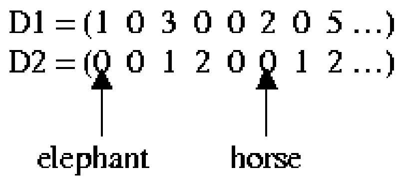


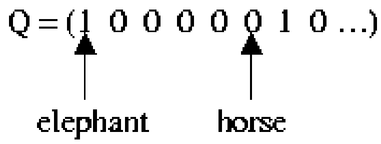
Figure 4. Partial compression of a treap

3 Vector Space Model

The vector space model is formal model of information retrieval. Vector space models rely on the assumption that the meaning of a document can be derived from the document's constituent terms. The idea is to model each document as a vector in which each term has a weight. Similarly, the query is also represented as a vector. Under this model, distinct terms available in all documents form a vocabulary. A vocabulary of n terms forms an n-dimensional vector space, i.e. every distinct term is a basis in the vector space. Suppose there is a vocabulary is elephant, ..., horse, ... Then two example documents will look like:



Consequently, a query that requires that both "elephant" and "horse" appear in the document could be:



By modeling the documents and query as vectors in the same space, vector space model provides an elegant way of implementing different information retrieval tasks such as ranked keyword searches, relevance feedback, and semantic indexing . To speed up the search process, this model is often used in conjunction with clustering technique.

3.1 Selecting Weights

To represent a document in a vector space, we need to assign each term in the document with a weight. We define this weight as wd,t, i.e.

$$w_{d,t} = \text{weight of term } t \text{ in the document}$$

The simplest way is to define $w_{d,t}$ as a binary variable, i.e.

$$w_{d,t} = \begin{cases} 1 & \text{if term } t \text{ appears in document;} \\ 0 & \text{otherwise.} \end{cases}$$

The main drawback of this weighting is that it fails to consider the appearance frequency of a term in a document. However, if a term appears more often in on e document than others, it should be concluded that the document is more relevant to the term. Hence, if $f_{d,t}$ is the number of times that term t appears in document d, we have:

$$f_{d,t} = \text{frequency of term } t \text{ within document } d$$

$$w_{d,t} = f_{d,t}$$

A disadvantage of the above equation is that the amount of $f_{d,t}$ varies in a very large range. To avoid the situation that is $w_{d,t}$ too large, we can use the following:

$$w_{d,t} = log(1 + f_{d,t})$$

In this equation, $f_{d,t}$ is added with 1 to make $w_{d,t}$ equals 0 when term t does not appear in document d. Inspired by the information theory, we can derive the information content of term t as:

$$w_{d,t} = log(\frac{N}{f_t})$$

where f_t is the number of documents that contain term t. In fact, $\frac{f_t}{N}$ can be seen as the probability of term t in the collection of documents. Thus w_t indicates the discriminatory power of term t and is called the inversed document freq uency of term t. Accordingly, we can calculate $w_{d,t}$ as the log frequency scale d by the information content:

$$w_{d,t} = log(\frac{N}{f_t})log(1 + f_{d,t})$$

The idea of this weighting that a term occurs frequently in the document but rarely in the remaining of the collection of documents has a high weight. Thus the document contains the term is more relevant to the term.

3.2 Similarity between Vectors

Since now both documents and query are modeled as vectors, the querying process is to check the similarity between corresponding vectors. In the rest of this section, we use wd and wq to represent the vectors of document and query, respectively. An intuitive measure of similarity is the dot product or inner product , $w_d.w_q$. As illustrated in Figure 5, it evident that the more similar are w_d and w_q, the larger is the inner product.

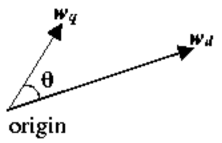


Figure 5. Similarity of two vectors

Euclidean distance is another familiar measure of closeness. However, it takes a large numerical value when two vectors are very different. Thus we take the reciprocal, which is the inverse Euclidean distance between the two vectors:

$$\frac{1}{||w_d - w_t||}$$

where $||.||$ is the Euclidean norm of a vector.

The problem of the above two measures is that both weight longer documents more heavily. To avoid this problem we can take a normalization of the inner product , which gives the Cosine measure:

$$\frac{w_d.w_q}{||w_d||.||w_t||}$$

The above measure gets its name because $w_d.w_q = ||w_d||.||w_d||cos\theta$. In other words, proximity is measured by the cosine of the angle between the two vectors . Because the angle is independent of the vector lengths, a longer document will not over emphasize the proximity measure.

3.3 Querying using the Cosine Measure and Inverted Lists

Directly calculated Cosine measure from query to every document in a large coll ection is potentially expensive in terms of CPU time and disk storage. To reduce the computation cost, we can save term frequency within a document along with the document number in the lexicon. In other words, we store (document, frequency) pairs using an inverted list for a term t as follows:

$$< t; w + t; ((d_{t,1}, f_{d_{t,1},t}), (d_{t,2}, f_{d_{t,2},t}), \ldots, (d_{t,m}, f_{d_{t,m},t})) >$$

where w_t is inverted document frequency of t, $d_{t,i}$ is the ith document that con tains term i, and $f_{d_{t,i},t}$ is the occurrence frequency of term t within the ith document. For example, an invert list for term apple could be:

$$< apple; .8; ((3,3), (6,1), (29,6)) >$$

The inverted lists have to be compressed for efficient storage. Typically frequency counts can be compressed at least as well as distances. In the TREC project , 2 bits per pointer can be achieved.

3.4 Searching

Based on the techniques developed before, in Figure 6 we list the algorithm of searching query Q in the document collection . In this algorithm, searching results will be stored in an accumulator set A, which is initialized to be empty at line 2. Then the algorithm searches each term in the query. For each term t, a lexicon search will return the corresponding inverted list P_t, which is de-compressed at line 5. Next an inner loop transverse each entry in the inverted list to calculate the similarity measure. For each entry, the product of w_t and $f_{d,t}$ is calculated. This product is the contribution of the term t's dimension to the inner product. Next the partial sum of inner product is updated. If the corresponding document is not included in the accumulator set yet, this document has to be added because term t occurs in it. After the loop begins at line 3, all documents containing terms in the query are found and their inner product with the query are stored in A. To derive the similarity measure for a document d in A, its inner product item is normali zed by its norm W_d (Normalization by the norm of Q is unnecessary because it is a constant factor for all documents in A.). To avoid the situation that the size of A can get large, A can be pruned so only top K (k documents are kept during search.

```
Algorithm: Search(Q)
1   A = Φ                        (Accumulators)
2   For each term t ∈ Q
3       <t; w_t; P_t> = Search lexicon for t
4       P_t = uncompress(P_t)
5       For each (d, f_d,t) in P_t
6           if a_d ∈ A
7               a_d = a_d + w_t f_d,t
8           else
9               a_d = w_t f_d,t
10              A = A + { a_d }
11  For each a_d ∈ A
12  a_d = a_d / W_d
13  Select k documents from A with largest a_d
```

Figure 6. Querying algorithm

3.5 Relevance Feedback

Suppose a query is submitted and some documents are returned. The user checks the results and indicates the relevance or irrelevance of them. Then another query can be launched to get more accurate search results. This process is called iterative retrieval guided by relevance feedback. The vector space model provides an elegant way to implement relevance feedback. Consider a sequence of queries $Q_1, Q_2, \ldots, Q_m$ in which R_i and I_i are the releva nt and irrelevant documents returned by query i. Typically R_i and I_i are marked by the user. Query $i+1$ can be updated by the feedback information. The simplest updating strategy is the **Dec Hi** Strategy:

$$Q_{i+1} = Q_i + \sum_{d \in R_i} D_d - D_n$$

where D_n is the most irrelevant document. This strategy compensates current query by the direction of summed vectors of all relevant documents. Also new query , Q_{i+1}, is adjusted to be far away from the direction of most irrelevant document. An example is given in Figure 7. After query i, it is indicated that document R_{i1} and R_{i2} are relevant, while I_{i1} is irrelevant. Thus the query is updated acc ordingly, which is shown in the right part of Figure 7.

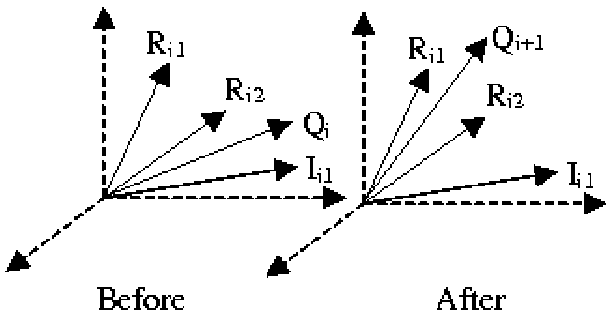


Figure 7. Relevance feedback in vector space

More generally, a new query can be derived by taking into account all results found in the current query as well as the initial and current queries:

$$Q_{i+1} = \pi Q_0 + \omega Q_i + \alpha \sum_{d \in R_i} D_d - \beta \sum_{d \in I_i} D_d$$

where π, ω, α, and β are all positive weighting constants. To efficiently implement relevance feedback, it is often to set an upper bound on the number the number of nonzero entries in each iteration of Q_i, or use clustering technique for the searches.

References

[1] Dan Blandford and Guy Blelloch, Functional Set Operations with Treaps, 2001.

[2] Alistair Moffat, Ola Petersson, Nicholas C. Wormald, A Tree Based Mergesort, *Acta Informatica* 35(9):775-793, 1998

[3] H. Kaplan and R. Tarjan, Persistent Lists with Catenation via Recursive Slow-Down, In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93-102, 1993

[4] H. Kaplan and R. Tarjan, Purely functional representations of catenable sorted lists. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 202-211 1996