Search Engines

Information Retrieval in Practice

Indexes

- Arrays
- Hash table
- Queue
- Priority Queue
- B-trees

unsorted arrays are slow to search, and sorted arrays are slow at insertion. By contrast, hash tables and trees are fast for both search and insertion.

Indexes

- Arrays
- Hash table
- Queue
- Priority Queue
- B-trees

Which one is good for Text Search?

Indexes

- Arrays
- Hash table
- Queue
- Priority Queue
- B-trees

Which one is good for Text Search?

Indexes

- **Which one is good for Text Search?**
- **Efficient query processing is a particularly important problem in web search.**
- **The query processing algorithm depends on the retrieval model, and dictates the contents of the index.**

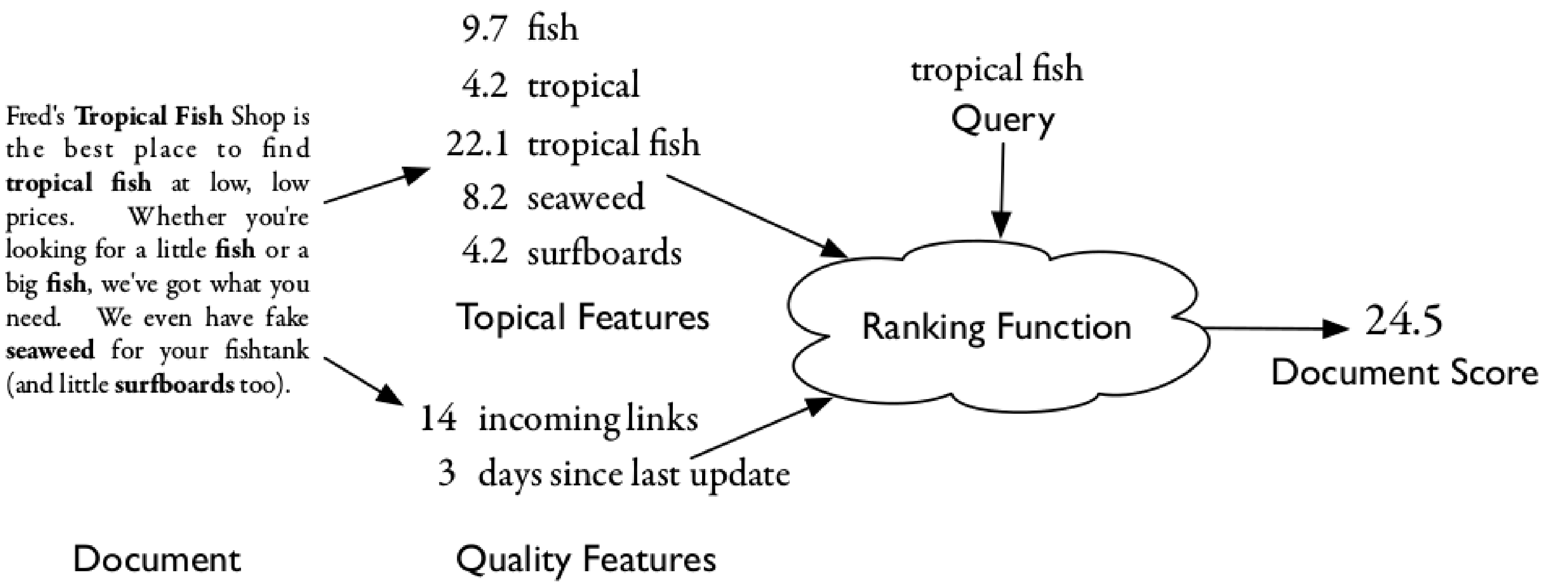
RANKING

Text search engines use a particular form of search: *ranking*

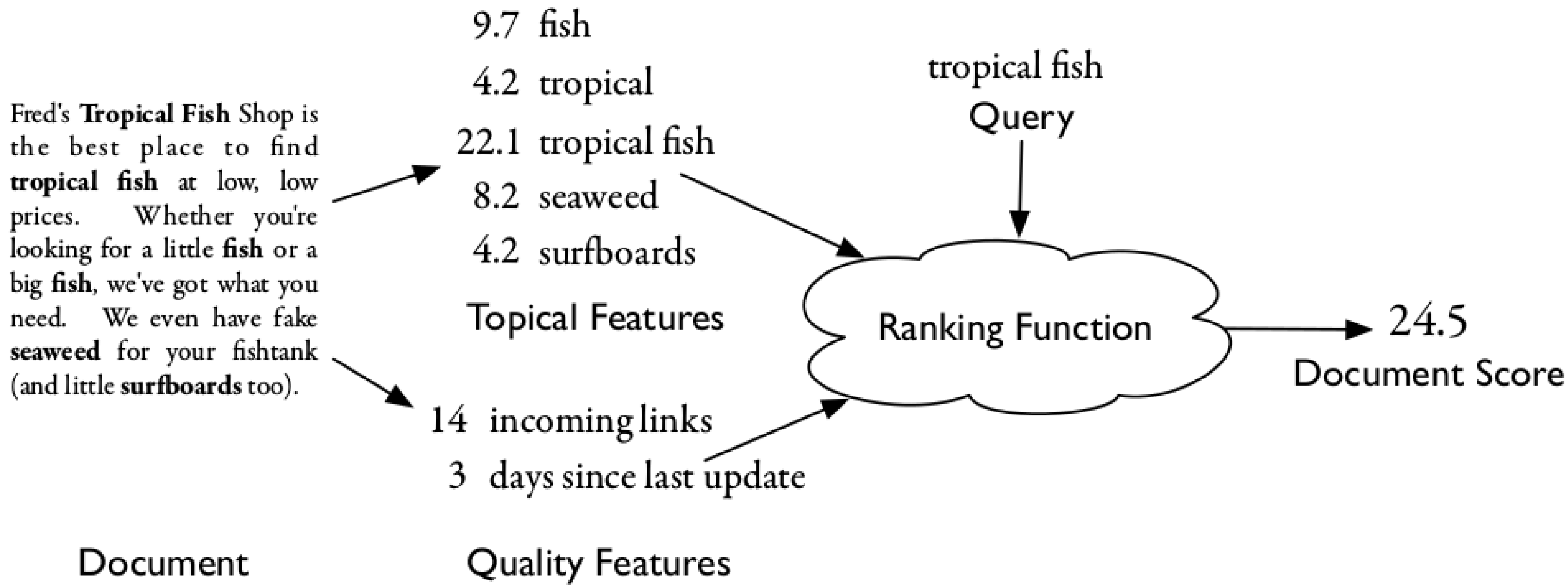
- documents are retrieved in sorted order according to a score computing using the document representation, the query, and a *ranking algorithm*

- What is a reasonable abstract model for ranking?
  - enables discussion of indexes without details of retrieval model

Abstract Model of Ranking



Abstract Model of Ranking

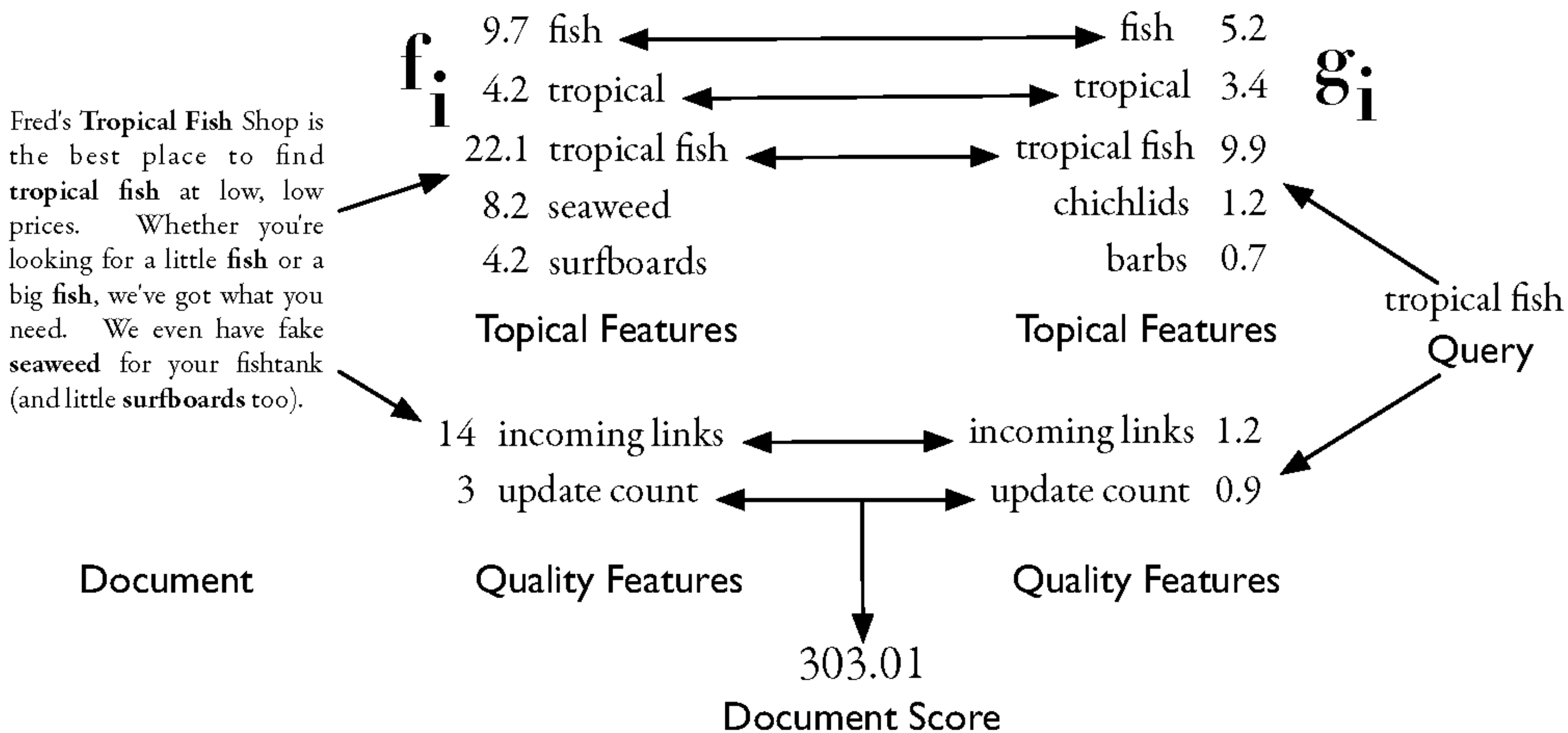


1. The text is transformed into index terms or document features
2. Topical features estimate the degree to which the document is about a particular subject.
3. Document quality feature:
    3.1 The number of web pages that link to this document,
    3.2 The number of days since this page was last updated.

More Concrete Model

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

f_i is a document feature function
g_i is a query feature function



Inverted Index

- Each index term is associated with an *inverted list*
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

Example “Collection”

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index

and	1
aquarium	3
are	3 4
around	1
as	2
both	1
bright	3
coloration	3 4
derives	4
due	3
environments	1
fish	1 2 3 4
fishkeepers	2
found	1
fresh	2
freshwater	1 4
from	4
generally	4
in	1 4
include	1
including	1
iridescence	4
marine	2
often	2 3
only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1 4
saltwater	2
species	1
term	2
the	1 2
their	3
this	4
those	2
to	2 3
tropical	1 2 3
typically	4
use	2
water	1 2 4
while	4
with	2
world	1

Inverted Inde with counts

- supports better ranking algorith

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

Inverted Index with positions

- supports proximity matches

Term	Postings
and	1,15
aquarium	3,5
are	3,3 4,14
around	1,9
as	2,21
both	1,13
bright	3,11
coloration	3,12 4,5
derives	4,7
due	3,7
environments	1,8
fish	1,2 1,4 2,7 2,18 2,23 3,2 3,6 4,3 4,13
fishkeepers	2,1
found	1,5
fresh	2,13
freshwater	1,14 4,2
from	4,8
generally	4,15
in	1,6 4,1
include	1,3
including	1,12
iridescence	4,9
marine	2,22
often	2,2 3,10
only	2,10
pigmented	4,16
popular	3,4
refer	2,9
referred	2,19
requiring	2,12
salt	1,16 4,11
saltwater	2,16
species	1,18
term	2,5
the	1,10 2,4
their	3,9
this	4,4
those	2,11
to	2,8 2,20 3,8
tropical	1,1 1,7 2,6 2,17 3,1
typically	4,6
use	2,3
water	1,17 2,14 4,12
while	4,10
with	2,15
world	1,11

Proximity Matches

- Matching phrases or words within a window
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- Word positions in inverted lists make these types of query features efficient
  - e.g.,

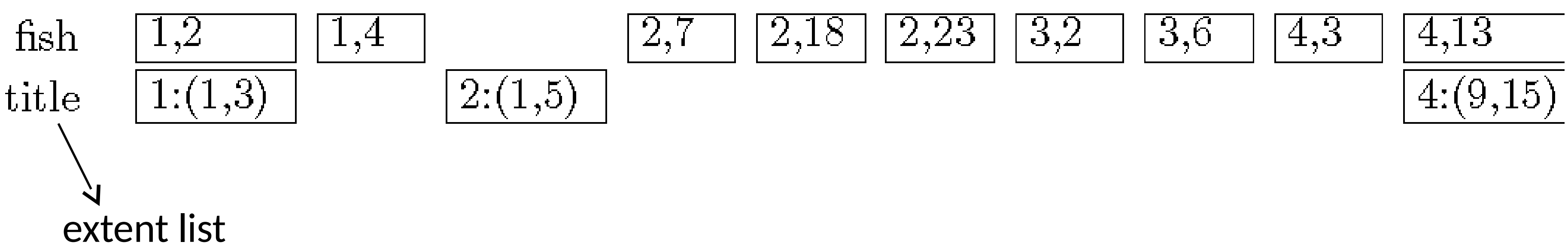
tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

Fields and Extents

- Document structure is useful in search
  - *field* restrictions
    - e.g., date, from:, etc.
  - some fields more important
    - e.g., title
- Options:
  - separate inverted lists for each field type
  - add information about fields to postings
  - use *extent lists*

Extent Lists

- An *extent* is a contiguous region of a document
  - represent extents using word positions
  - inverted list records all extents for a given field type
  - e.g.,



Other Issues

- Precomputed scores in inverted list
  - e.g., list for "fish" [(1:3.6), (3:2.2)], where 3.6 is total feature value for document 1
  - improves speed but reduces flexibility
- Score-ordered lists
  - query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries

Compression

- Inverted lists are very large
  - e.g., 25-50% of collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good *compression ratios* and are easy to decompress
- *Lossless* compression – no information lost

Skipping

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 doc-

ument order, there is a very simple algorithm for processing this query:

Skipping

- 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 d g < d a , set d g to the next document number in the “galago” list.

– If d a < d g , set d a to the next document number in the “animal” list.

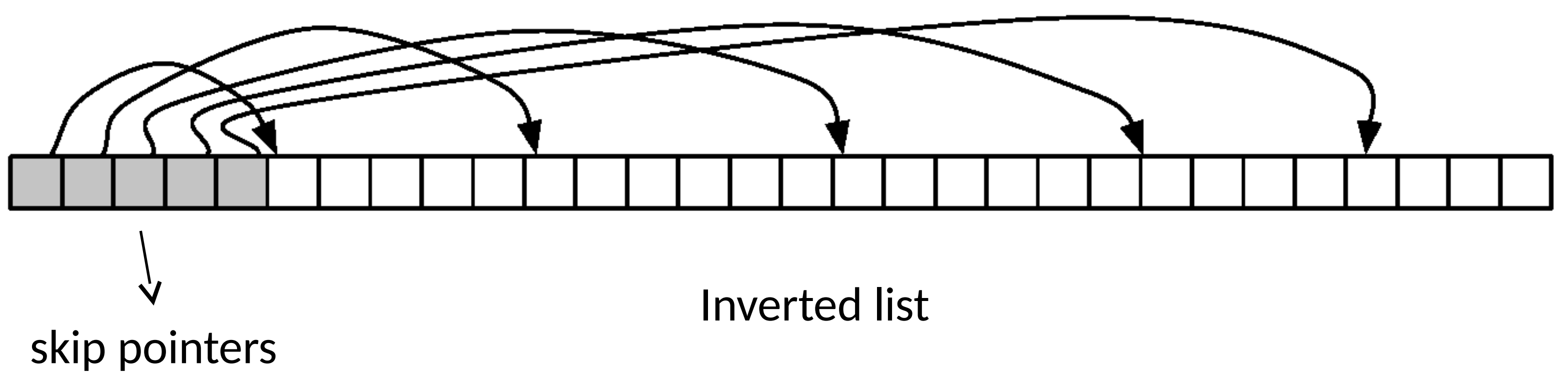
– If d a = d g , the document d a contains both “galago” and “animal”. Move both d g and d a to the next documents in the inverted lists for “galago” and “animal,” respectively.

Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be very inefficient
  - “Skipping” ahead to check document numbers is much better
  - Compression makes this difficult
    - Variable size, only d-gaps stored
- Skip pointers are additional data structure to support skipping

Skip Pointers

- A skip pointer (*d*, *p*) contains a document number *d* and a byte (or bit) position *p*
  - Means there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*



Index Construction

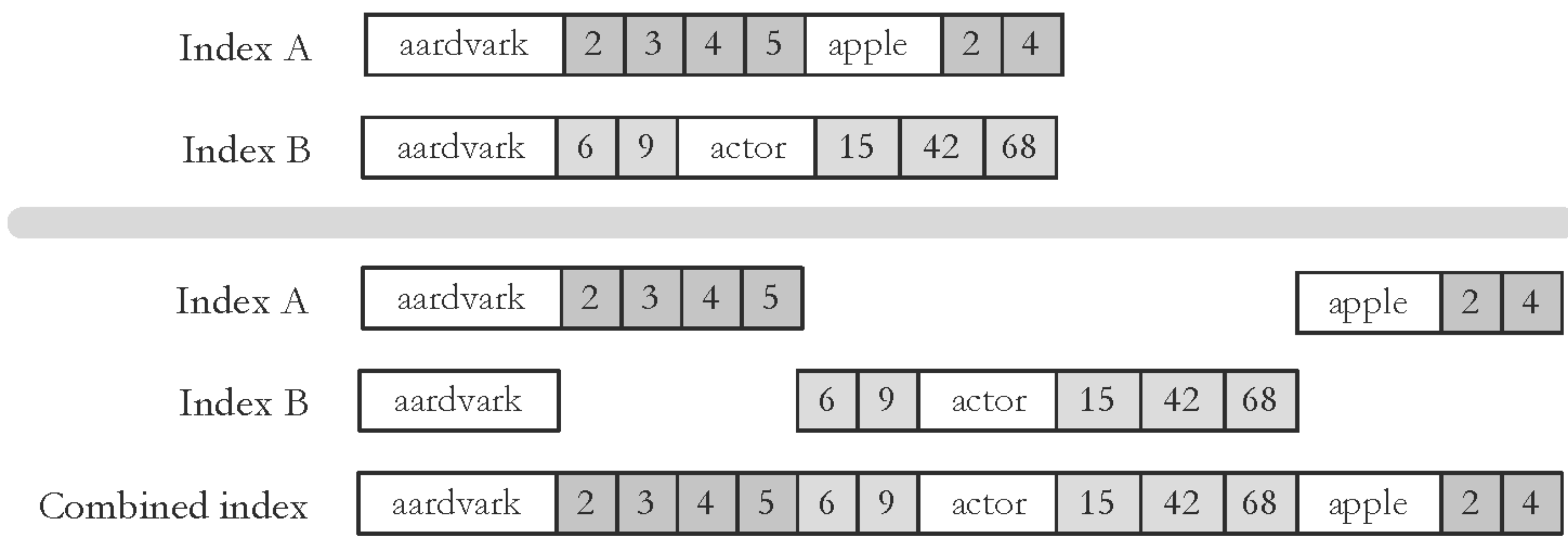
```
procedure BuildIndex(D)                         ▷ D is a set of text documents
    I ← HashTable()                             ▷ Inverted list storage
    n ← 0                                       ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                            ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

Simple in-memory indexer

Merging

- Merging addresses limited memory problem
  - Build the inverted list structure until memory runs out
  - Then write the partial index to disk, start making a new one
  - At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists must be designed so they can be merged in small pieces
  - e.g., storing in alphabetical order

Merging



Index A
aardvark 2 3 4 5 apple 2 4
Index B
aardvark 6 9 actor 15 42 68
Index A
aardvark 2 3 4 5
apple 2 4
Index B
aardvark
6 9 actor 15 42 68
Combined index
aardvark 2 3 4 5 6 9 actor 15 42 68 apple 2 4

Distributed Indexing

- Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)
- Large numbers of inexpensive servers used rather than larger, more expensive machines
- *MapReduce* is a distributed programming tool designed for indexing and analysis tasks

Example

- Given a large text file that contains data about credit card transactions
  - Each line of the file contains a credit card number and an amount of money
  - Determine the number of unique credit card numbers
- Could use hash table – memory problems
  - counting is simple with sorted file
- Similar with distributed approach
  - sorting and placement are crucial

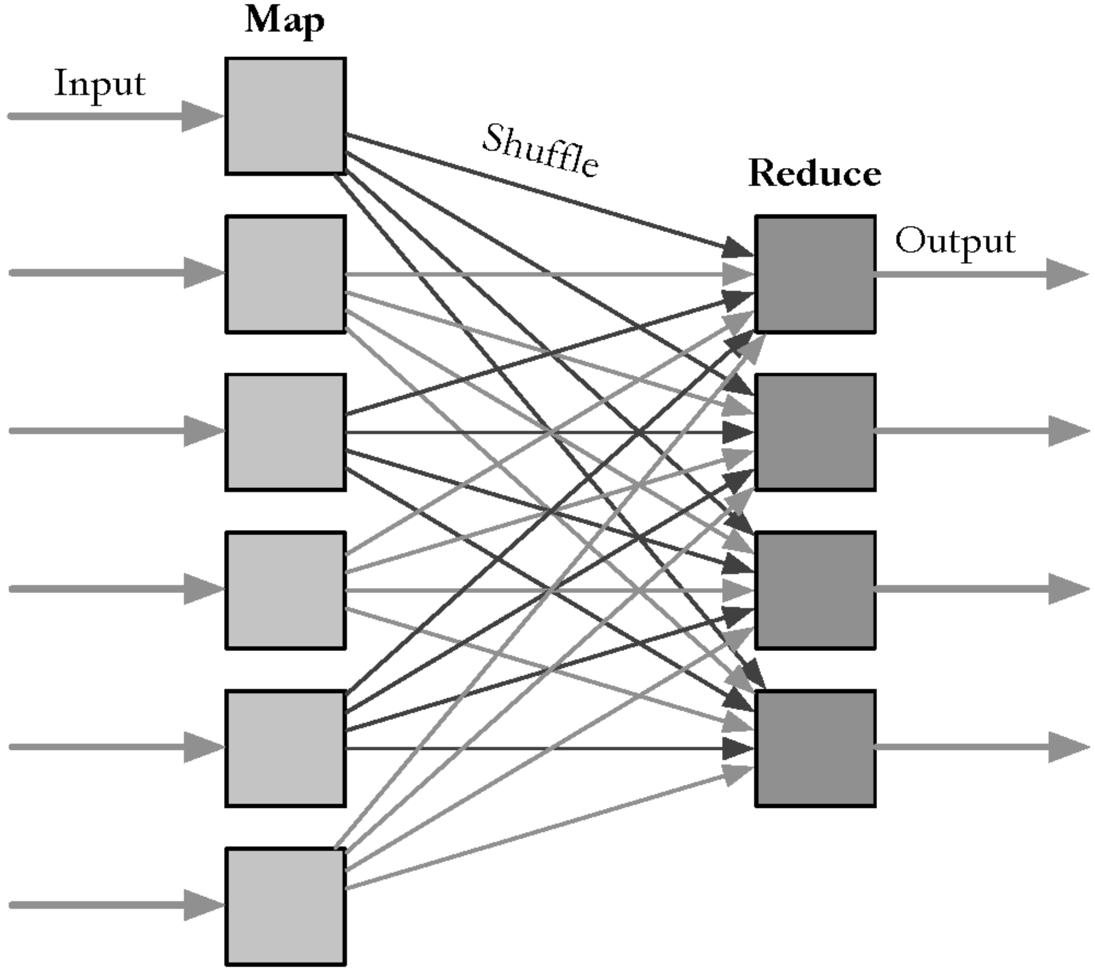
MapReduce

- Distributed programming framework that focuses on data placement and distribution
- *Mapper*
  - Generally, transforms a list of items into another list of items of the same length
- *Reducer*
  - Transforms a list of items into a single item
  - Definitions not so strict in terms of number of outputs
- Many mapper and reducer tasks on a cluster of machines

MapReduce

- Basic process
  - *Map* stage which transforms data records into pairs, each with a key and a value
  - *Shuffle* uses a hash function so that all pairs with the same key end up next to each other and on the same machine
  - *Reduce* stage processes records in batches, where all pairs with the same key are processed at the same time
- *Idempotence* of Mapper and Reducer provides fault tolerance
  - multiple operations on same input gives same output

MapReduce



Example

```
procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure

procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
```

Indexing Example

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, document:position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```

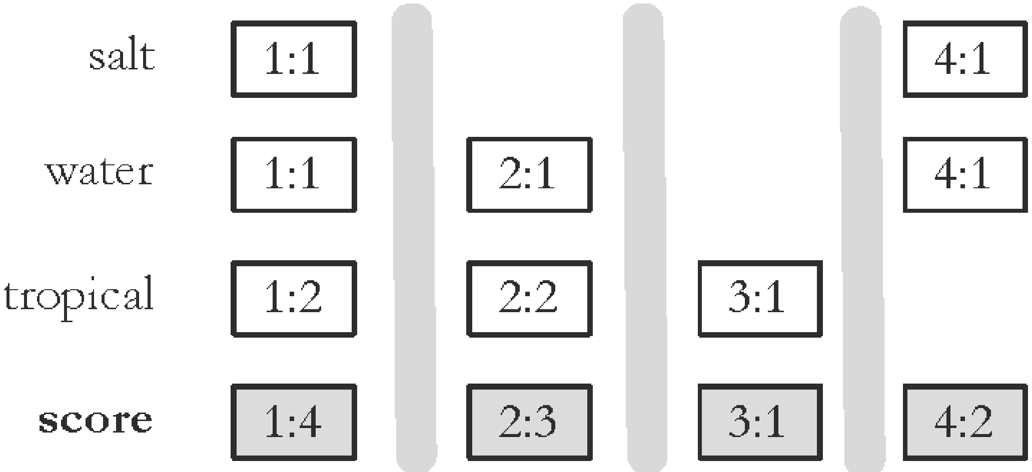
Result Merging

- Index merging is a good strategy for handling updates when they come in large batches
- For small updates this is very inefficient
  - instead, create separate index for new documents, merge *results* from both searches
  - could be in-memory, fast to update and search
- Deletions handled using *delete list*
  - Modifications done by putting old version on delete list, adding new version to new documents index

Query Processing

- Document-at-a-time
  – Calculates complete scores for documents by processing all term lists, one document at a time
- Term-at-a-time
  – Accumulates scores for documents by processing term lists one at a time
- Both approaches have optimization techniques that significantly reduce time required to generate scores

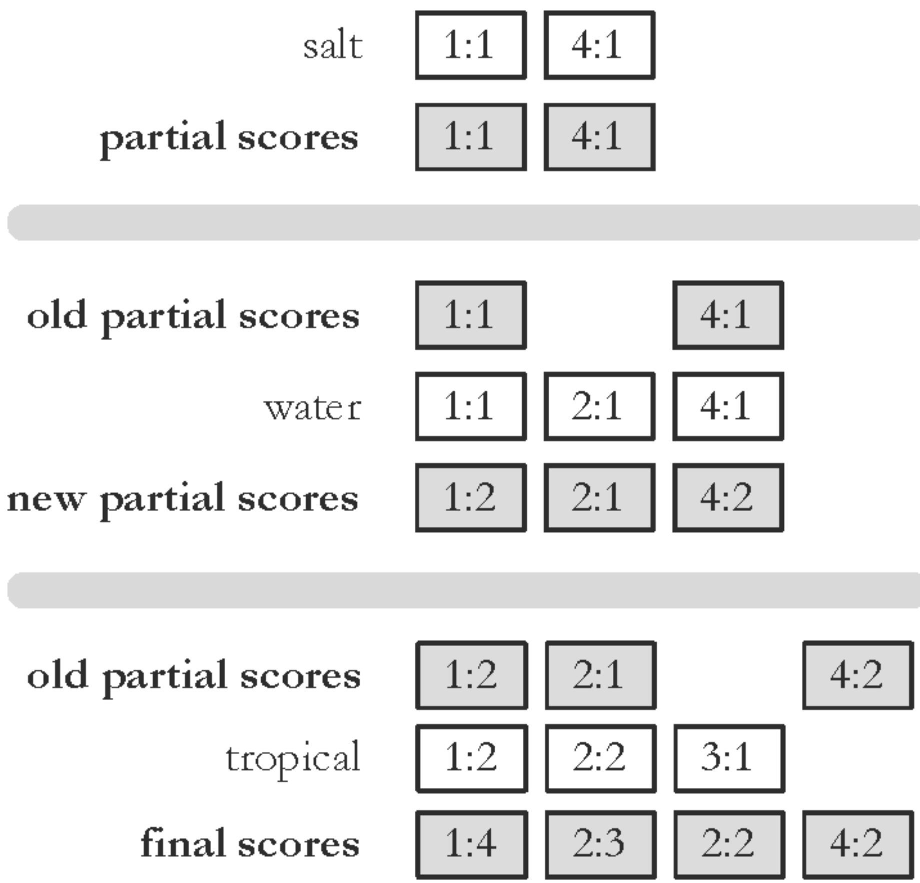
Document-At-A-Time



Document-At-A-Time

```
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
        for all inverted lists l_i in L do
            if l_i points to d then
                s_D ← s_D + g_i(Q)f_i(l_i)          ▷ Update the document score
                l_i.movePastDocument( d )
            end if
        end for
        R.add( s_D, D )
    end for
    return the top k results from R
end procedure
```

Term-At-A-Time



Term-At-A-Time

```
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                    ▷ Accumulator contains the document score
        R.add( s_D, D )
    end for
    return the top k results from R
end procedure
```

Optimization Techniques

- Term-at-a-time uses more memory for accumulators, but accesses disk more efficiently
- Two classes of optimization
  - Read less data from inverted lists
    - e.g., skip lists
    - better for simple feature functions
  - Calculate scores for fewer documents
    - e.g., conjunctive processing
    - better for complex feature functions

Conjunctive Term-at-a-Time

```
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
            if i = 0 then
                d ← l_i.getCurrentDocument()
                A_d ← A_d + g_i(Q)f(l_i)
            else
                d ← l_i.getCurrentDocument()
                d ← A.getNextDocumentAfter(d)
                l_i.skipForwardTo(d)
                if l_i.getCurrentDocument() = d then
                    A_d ← A_d + g_i(Q)f(l_i)
                else
                    A.remove(d)
                end if
            end if
        end while
    end for
    for all accumulators A_d in A do
        s_D ← A_d                  ▷ Accumulator contains the document score
        R.add( s_D, D )
    end for
    return the top k results from R
end procedure
```

Conjunctive Document-at-a-Time

```
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
    while all lists in L are not finished do
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() > d then
                d ← l_i.getCurrentDocument()
            end if
        end for
        for all inverted lists l_i in L do l_i.skipForwardToDocument(d)
            if l_i points to d then
                s_d ← s_d + g_i(Q)f_i(l_i)          ▷ Update the document score
                l_i.movePastDocument( d )
            else
                break
            end if
        end for
        R.add( s_d, d )
    end while
    return the top k results from R
end procedure
```

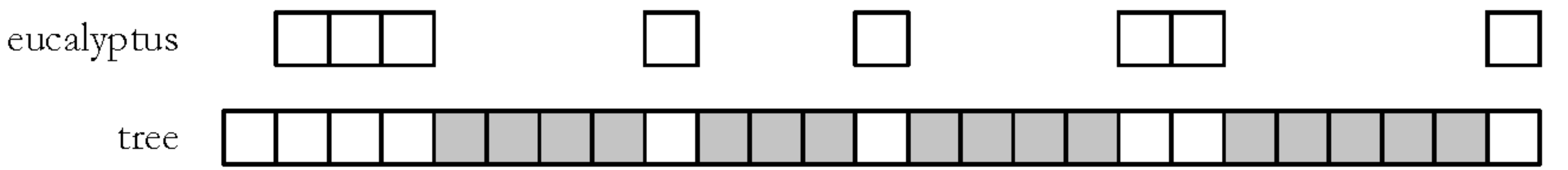
Threshold Methods

- Threshold methods use number of top-ranked documents needed (*k*) to optimize query processing
  - for most applications, *k* is small
- For any query, there is a *minimum score* that each document needs to reach before it can be shown to the user
  - score of the *k*th-highest scoring document
  - gives *threshold* τ
  - optimization methods estimate τ' to ignore documents

Threshold Methods

- For document-at-a-time processing, use score of lowest-ranked document so far for τ'
  - for term-at-a-time, have to use k_{th}-largest score in the accumulator table
- *MaxScore* method compares the maximum score that remaining documents could have to τ'
  - *safe* optimization in that ranking will be the same without optimization

MaxScore Example



- Indexer computes μ_{tree}
  - maximum score for any document containing just "tree"
- Assume k =3, τ' is lowest score after first three docs
- Likely that $\tau' > \mu_{tree}$
  - τ' is the score of a document that contains both query terms
- Can safely skip over all gray postings

Other Approaches

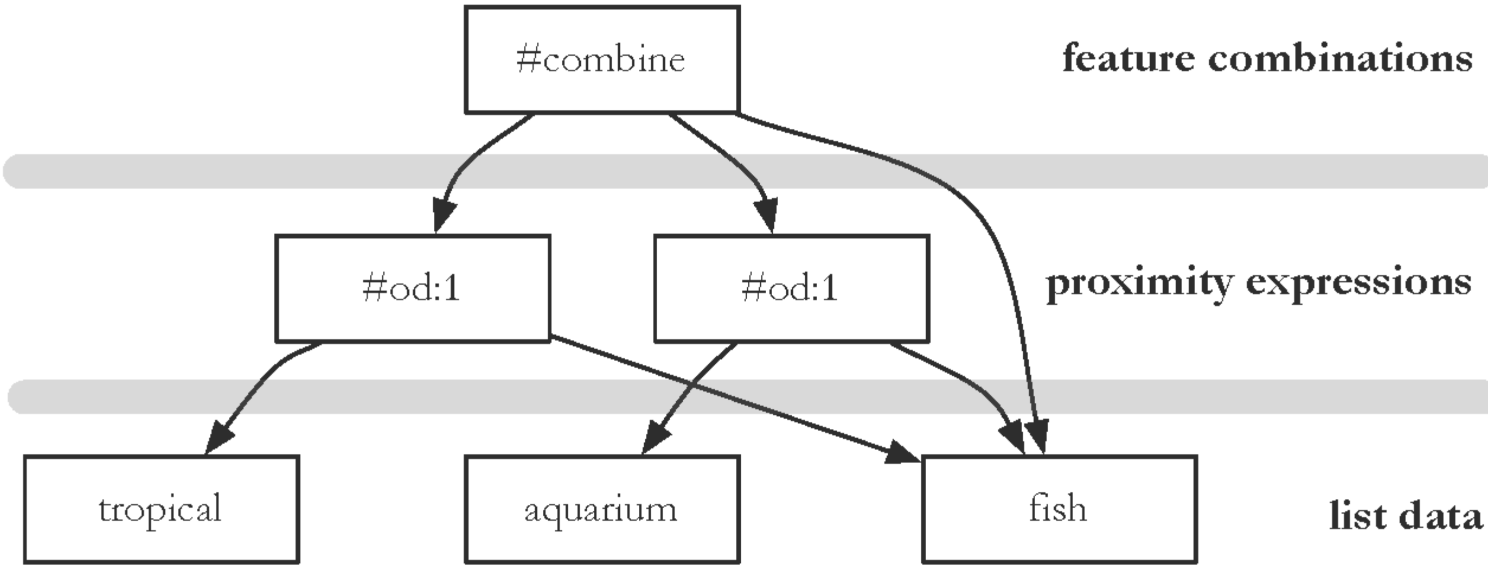
- Early termination of query processing
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization
- List ordering
  - order inverted lists by quality metric (e.g., PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to produce good documents

Structured Queries

- *Query language* can support specification of complex features
  - similar to SQL for database systems
  - *query translator* converts the user's input into the structured query representation
  - Galago query language is the example used here
  - e.g., Galago query:

#combine(#od:1(tropical fish) #od:1(aquarium fish) fish)

Evaluation Tree for Structured Query



Distributed Evaluation

- Basic process
  - All queries sent to a *director machine*
  - Director then sends messages to many *index servers*
  - Each index server does some portion of the query processing
  - Director organizes the results and returns them to the user
- Two main approaches
  - Document distribution
    - by far the most popular
  - Term distribution

Distributed Evaluation

- Document distribution
  - each index server acts as a search engine for a small fraction of the total collection
  - director sends a copy of the query to each of the index servers, each of which returns the top-*k* results
  - results are merged into a single ranked list by the director
- Collection statistics should be shared for effective ranking

Distributed Evaluation

- Term distribution
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - One of the index servers is chosen to process the query
    - usually the one holding the longest inverted list
  - Other index servers send information to that server
  - Final results sent to director

Caching

- Query distributions similar to Zipf
  - About ½ each day are unique, but some are very popular
- Caching can significantly improve effectiveness
  - Cache popular query results
  - Cache common inverted lists
- Inverted list caching can help with unique queries
- Cache must be refreshed to prevent stale data