Chapter 5

Ranking with Indexes
Indexes and Ranking

- Indexes are designed to support search
  - Faster response time, supports updates

- Text search engines use a particular form of search: ranking
  - Docs are retrieved in sorted order according to a score computed using the doc representation, the query, and a ranking algorithm

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

Fred's Tropical Fish Shop is the best place to find tropical fish at low, low prices. Whether you're looking for a little fish or a big fish, we've got what you need. We even have fake seaweed for your fishtank (and little surfboards too).
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
aquarium
are
around
as
both
bright
coloration
derives
due
environments
fish
fishkeepers
found
fresh
freshwater
from
generally
in
include
including
iridescence
marine
often

only
pigmented
popular
refer
referred
requiring
salt
saltwater
species
term
the
their
this
those
to
tropical
typically
use
water
while
with
world

posting
Inverted Index with counts - supports better ranking algorithms

<table>
<thead>
<tr>
<th>Word</th>
<th>Doc #</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>aquarium</td>
<td>3:1</td>
<td></td>
</tr>
<tr>
<td>are</td>
<td>3:1</td>
<td>4:1</td>
</tr>
<tr>
<td>around</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>as</td>
<td>2:1</td>
<td></td>
</tr>
<tr>
<td>both</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>bright</td>
<td>3:1</td>
<td></td>
</tr>
<tr>
<td>coloration</td>
<td>3:1</td>
<td>4:1</td>
</tr>
<tr>
<td>derives</td>
<td>4:1</td>
<td></td>
</tr>
<tr>
<td>due</td>
<td>3:1</td>
<td></td>
</tr>
<tr>
<td>environments</td>
<td>1:1</td>
<td>2:3</td>
</tr>
<tr>
<td>fish</td>
<td>1:2</td>
<td>2:3</td>
</tr>
<tr>
<td>fishkeepers</td>
<td>2:1</td>
<td></td>
</tr>
<tr>
<td>found</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>fresh</td>
<td>2:1</td>
<td></td>
</tr>
<tr>
<td>freshwater</td>
<td>1:1</td>
<td>4:1</td>
</tr>
<tr>
<td>from</td>
<td>4:1</td>
<td></td>
</tr>
<tr>
<td>generally</td>
<td>4:1</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>1:1</td>
<td>4:1</td>
</tr>
<tr>
<td>include</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>including</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>iridescence</td>
<td>4:1</td>
<td></td>
</tr>
<tr>
<td>marine</td>
<td>2:1</td>
<td></td>
</tr>
<tr>
<td>often</td>
<td>2:1</td>
<td>3:1</td>
</tr>
<tr>
<td>only</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>pigmented</td>
<td></td>
<td>4:1</td>
</tr>
<tr>
<td>popular</td>
<td></td>
<td>3:1</td>
</tr>
<tr>
<td>refer</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>referred</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>requiring</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>salt</td>
<td></td>
<td>1:1</td>
</tr>
<tr>
<td>saltwater</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>species</td>
<td></td>
<td>1:1</td>
</tr>
<tr>
<td>term</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>the</td>
<td></td>
<td>1:1</td>
</tr>
<tr>
<td>their</td>
<td></td>
<td>3:1</td>
</tr>
<tr>
<td>this</td>
<td></td>
<td>4:1</td>
</tr>
<tr>
<td>those</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>to</td>
<td></td>
<td>2:2</td>
</tr>
<tr>
<td>tropical</td>
<td></td>
<td>1:2</td>
</tr>
<tr>
<td>typically</td>
<td></td>
<td>4:1</td>
</tr>
<tr>
<td>use</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>water</td>
<td></td>
<td>1:1</td>
</tr>
<tr>
<td>while</td>
<td></td>
<td>4:1</td>
</tr>
<tr>
<td>with</td>
<td></td>
<td>2:1</td>
</tr>
<tr>
<td>world</td>
<td></td>
<td>1:1</td>
</tr>
</tbody>
</table>

No. of time the word occurs
Inverted Index with Positions - Supports Proximity Matches

- 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

Position in the doc
Doc #
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
```
Indexing

- **Dense Index**: For every unique search-key value, there is an index record.

- **Sparse Index**: Index records are created for some search-key values.
  - Sparse index is slower, but requires less space & overhead.

- **Primary Index**: Defined on an ordered data file, ordered on a search key field & is usually the primary key.
  - A sequentially ordered file with a primary index is called index-sequential file.
  - A binary search on the index yields a pointer to the record.
  - Index value is the search-key value of the first data record in the block.
Figure. Dense index

Figure. Sparse index
Figure. Primary index on the ordering key field of a file
Multi-Level Indices

- **Leaf-node level**: pointers to the original data file
- **First-level index**: pointers to the original index file
- **Second-level index**: primary index to the original index file
- **Third-level index**: forms the index of the 2\textsuperscript{nd}-level
- (Rare) **fourth-level index**: top level index (fit in one disk block)
- Form a search tree, such as B\textsuperscript{+}-tree structures
- Insertion/deletion of new indexes are not trivial in indexed files
Figure. A two-level primary index
B⁺-Tree (Multi-level) Indices

- Frequently used index structure
- Allow efficient insertion/deletion of new/existing search-key values
- A *balanced tree structure*: all leaf nodes are at the same level (which form a *dense index*)
- Each node, corresponding to a disk block, has the format:

  \[
  \begin{array}{cccccc}
  P_1 & K_1 & P_2 & \cdots & P_{n-1} & K_{n-1} & P_n \\
  \end{array}
  \]

  where \( P_i, 1 \leq i \leq n \), is a pointer
  \( K_i, 1 \leq i \leq n-1 \), is a search-key value &
  \( K_i < K_j, i < j \), i.e., search-key values are in order

  \[
  \begin{array}{cccccc}
  P_1 & K_1 & \cdots & K_{i-1} & P_i & K_i & \cdots & K_{n-1} & P_n \\
  \end{array}
  \]

  - In each leaf node, \( P_i \) points to either (i) a document with search-key value \( K_i \) or (ii) a bucket of pointers, each points to a document with search-key value \( K_i \)
B+-Tree (Multi-level) Indices

- Each leaf node is kept between half full & completely full, i.e., \( \lceil (n-1)/2 \rceil, n-1 \) search-key values
- Non-leaf nodes form a sparse index
- Each non-leaf node (except the root) must have \( \lceil n/2 \rceil, n \) pointers
- No. of Block accesses required for searching a search-key value @ leaf-node level is \( \log \lceil n/2 \rceil(K) \)
  where \( K = \) no. of unique search-key values & \( n = \) no. of indices/node
- Insertion into a full node causes a split into two nodes which may propagate to higher tree levels
  
  Note: if there are \( n \) search-key values to be split, put the first \( \lceil (n-1)/2 \rceil \) in the existing node & the remaining in a new node
- A less than half full node caused by a deletion must be merged with neighboring nodes
**B⁺-Tree Algorithms**

- **Algorithm 1.** Searching for a record with search-key value $K$, using a B⁺-Tree.

Begin

$n \leftarrow$ block containing root node of B⁺-Tree;
read block $n$;
while ($n$ is not a leaf node of the B⁺-Tree) do

begin

$q \leftarrow$ number of tree pointers in node $n$;
if $K < n.K_1$ /* $n.K_i$ refers to the $i^{th}$ search-key value in node $n$ */
then $n \leftarrow n.P_1$ /* $n.P_i$ refers to the $i^{th}$ pointer in node $n$ */
else if $K \geq n.K_{q-1}$
then $n \leftarrow n.P_q$
else begin

search node $n$ for an entry $i$ such that $n.K_{i-1} \leq K < n.K_i$;
$n \leftarrow n.P_i$;
end; /*ELSE*/

read block $n$;
end; /*WHILE*/

search block $n$ for entry $K_i$ with $K = K_i$ /*search leaf node*/
if found, then read data file block with address $P_i$ and retrieve record
else record with search-key value $K$ is not in the data file;
end. /*Algorithm 1*/
**B\(^+\)-Tree Algorithms**

**Algorithm 2.** Inserting a record with search-key value \( K \) in a B\(^+\)-Tree of order \( p \).

\[
/* A B^+\text{-Tree of order } p \text{ contains at most } p-1 \text{ values an } p \text{ pointers}*/
\]

Begin

\[ n \leftarrow \text{block containing root node of } B^+\text{-Tree} ; \]
read block \( n \);
set stack \( S \) to empty;
while (\( n \) is not a leaf node of the \( B^+\)-Tree ) do
begin
push address of \( n \) on stack \( S \); /* \( S \) holds parent nodes that are needed in case of split */
\[ q \leftarrow \text{number of tree pointers in node } n ; \]
if \( K < n.K_1 \) /* \( n.K_i \) refers to the \( i \)th search-key value in node \( n \) */
then \( n \leftarrow n.P_1 \) /* \( n.P_i \) refers to the \( i \)th pointer in node \( n \) */
else if \( K \geq n.K_{q-1} \)
then \( n \leftarrow n.P_q \)
else begin
search node \( n \) for an entry \( i \) such that \( n.K_{i-1} \leq K < n.K_i ; \)
\[ n \leftarrow n.P_i ; \]
end; /* ELSE */
read block \( n \);
end; /* WHILE */
search block \( n \) for entry \( K_i \) with \( K = K_i ; /* search leaf node */

Algorithm 2 Continue

if found
  then return /*record already in index file - no insertion is needed */
else
  begin /* insert entry in B+-Tree to point to record */
    create entry \((P, K)\), where \(P\) points to file block containing new record;
    if leaf node \(n\) is not full
      then insert entry \((P, K)\) in correct position in leaf node \(n\)
    else
      begin /* leaf node \(n\) is full – split */
        copy \(n\) to temp; /* temp is an oversize leaf node to hold extra entry */
        insert entry \((P, K)\) in temp in correct position; /* temp now holds \(p+1\) entries of the form \((P_i, K_i)\) */
        new \(\leftarrow\) a new empty leaf node for the tree;
        \(j \leftarrow \lceil p/2 \rceil\)
        \(n \leftarrow\) first \(j\) entries in temp (up to entry \((P_j, K_j)\));
        \(n.P_{next} \leftarrow\) new; /* \(P_{next}\) points to the next leaf node*/
        new \(\leftarrow\) remaining entries in temp;
        \(K \leftarrow K_{j+1};\)
        /* Now we must move \((K, new)\) and insert in parent internal node. However, if parent is full, split may propagate */
      finished \(\leftarrow\) false;
  end
Algorithm 2 continue

Repeat
  if stack $S$ is empty, then /*no parent node*/
    begin /* new root node is created for the B$^+$-Tree */
      $root \leftarrow$ a new empty internal node for the tree;
      $root \leftarrow <n, K, new>$; /* set $P_1$ to $n$ & $P_2$ to new */
      finished $\leftarrow$ true;
    end
  else
    begin
      $n \leftarrow$ pop stack $S$;
      if internal node $n$ is not full, then
        begin /* parent node not full - no split */
          insert ($K$, $new$) in correct position in internal node $n$;
          finished $\leftarrow$ true
        end
      else
        else
          end
begin /* internal node $n$ is full with $p$ tree pointers – split */
    copy $n$ to $temp$; /* $temp$ is an oversize internal node */
    insert ($K$, $new$) in $temp$ in correct position; /* $temp$ has $p+1$ tree pointers */
    $new \leftarrow$ a new empty internal node for the tree;
    $j \leftarrow \lceil (p + 1)/2 \rceil$
    $n \leftarrow$ entries up to tree pointer $P_j$ in $temp$;
    /* $n$ contains $<P_1, K_1, P_2, K_2, \ldots, P_{j-1}, K_{j-1}, P_j>$ */
    $new \leftarrow$ entries from tree pointer $P_{j+1}$ in $temp$;
    /* $new$ contains $<P_{j+1}, K_{j+1}, \ldots, P_{p-1}, K_p, P_p, P_{p+1}>$ */
    $K \leftarrow K_j$
    /* now we must move ($K$, $new$) and insert in parent internal node */
end
end
until finished
end; /* ELSE */
end; /* ELSE */
end. /* Algorithm 2 */