Text Analytics

Indexing Terms

Ulf Leser
Searching Multiple Strings

• Often, we need to search for more than one string
  – Search all gene names from a dictionary in a given text
  – We want to search for the entire dictionary at once
• Let $P = \{P_1, P_2, \ldots, P_z\}$, $n = |P_1| + |P_2| + \ldots + |P_z|$
• First attempt
  – Z-box requires $O(m + |P_i|)$ for searching for pattern $P_i$
  – Naïve extension to $z$ patterns requires $O(z \cdot m + n)$
    • There is nothing we can save
• We shall improve this to $O(m + n + k)$
  – For same special definition of $k$ - later
Example

- $P = \{\text{banane, bohne, bohnern, wohnen, bohren}\}$
Constructing a Keyword Tree

• Complexity?

• **Construction is O(n)**
  
  - Start with $P_1$
  
  - Constructing the “tree” needs $O(|P_1|)$
  
  - Take $P_2$. Traverse the prefix of $P_2$ in the tree until
    
    • ... there is a mismatch at position $i$ in $P_2$. Insert a fork and create the branch for the rest of $P_2$ in $O(|P_2|-i)$
    
    • ... $P_2$ was matched completely
    
    • This needs $O(|P_2|)$ in either case
  
  - Repeat for $P_3$ – $P_z$

• Since paths are unique, there is no backtracking etc.
Failure Links

• Definition

Let $K$ be the Keyword Tree for a set $P$ of pattern. Let $k$ be a node of $K$

- Let $\text{length}(k)$ be the length of the longest true suffix of $\text{label}(k)$ which is also a prefix of any pattern in $P$
  - If suffix with length $>0$ exists, set $\text{length}(k)=0$
- Let $\text{fl}(k)$ denote the node with:
  
  \[ \text{label}(\text{fl}(k)) = \text{label}(k)[|\text{label}(k)|-\text{length}(k)+1 .. |\text{label}(k)| ] \]
  - If $\text{length}(k)=0$, set $\text{fl}(k)=\text{root}$

• Remarks

- The link $(k, \text{fl}(k))$ is called the Failure Link for $k$
- $\text{label}(\text{fl}(k))$ exactly is the „longest true suffix“ of $\text{label}(k)$
- $\text{fl}(k)$ must be unique
Example

$P = \{\text{banane, nabe, abnahme, na, abgabe}\}$

FLs to root are not shown
Searching with Failure Links

- Assume we search at position $j$ in $T$
- We match substring $T[j..]$ in $K$
  - If there is a match
    - Traverse down that match and set $j++$
    - If the reached node is marked, report the mark as match
  - If there is a mismatch at position $x$ in $T$
    - Let $k$ be the last match node
      - All children of $k$ are mismatches for $T[j+x]$
    - Follow the failure link of $k$ to node $fl(k)$
      - We have just seen $\text{label}(fl(k))$ in $T$
    - Continue matching at position $j+x$ in $T$ and node $fl(k)$ in $K$
  - If we reach a leaf $k$ at position $j+x-1$ in $T$
    - Report the mark of the leaf
    - Follow the failure link to node $fl(k)$
    - Continue matching at position $j+x$ in $T$ and node $fl(k)$ in $K$
But ...

\[ P = \{ \text{knabt, nabe, na} \} \]

\[ T = \text{knabenschaft} \]

- Algorithm matches KNAB in T
- B is the last matching symbol, failure link to NAB
- Proceed to NABE, report \( P_2 \)
- Follow fl to root and match on in T with NSCHAFT
- We missed \( P_3 \) (NA)!
  - Why?: \( P_2 \) contains \( P_3 \)
  - But hold on a second
Observation

- Let’s fix a node $k$
- We may reach all prefixes of any pattern which are identical to a suffix of $\text{label}(k)$ by following failure links
  - The longest such prefix is $fl(k)$
  - The others are $fl(fl(k))$, $fl(fl(...))$

$P=\{\text{knabe, nabe, abe, bele}\}$
Our Problematic Case

- Patterns containing other patterns
- Solution: We construct another set of pointers called **Output Links**
- Observation
  - Let $P_1$ be contained in $P_2$
  - Then $P_1$ must be the suffix of a prefix $P_2[1..i]$ for some $i \geq |P_1|$.
  - If $P_1$ is the longest prefix ($n P$) of $P_2[1..i]$, then $fl(P_2[i]) = P_1$
    - Which doesn’t help – usually, we will not follow this link during search
  - If this is not the case, there must exist a $P'$ with
    - $P'$ is the longest suffix of $P_2[1..i]$
    - Thus, $fl(P_2[i]) = P'$
    - Again: $P_1$ is suffix of $P'$ – but is it the longest?
      - Search recursive using failure links
      - Eventually, we must reach $P_1$
Example

$P = \{\text{knabe, na}\}$

$P = \{\text{eknabe, na, kna}\}$
Complexity

• During search
  - Let \( k \) be the number of matches of all patterns
  - The inner WHILE-loop is passed at most \( k \) times
  - Thus: \( O(m+k) \)

• Overall complexity
  - Build the keyword tree for \( P \) \( O(n) \) (trivial)
  - Compute failure links \( O(n) \) (BF)
    - This includes the output links
  - Search \( O(m+k) \)

• Total: \( O(n+m+k) \)
Content of this Lecture

- Inverted Files
  - Phrase and proximity search
  - Using a RDBMS
- Signature Files
  - S-Trees
Full-Text Indexing

• The fundamental operation in all our IR models: \texttt{find( q, D)}
  – Given a term, find all docs containing the term
  – Or: Given a set of terms, find all docs containing at least of them

• Can be implemented using online search
  – Boyer-Moore, Keyword-Trees, etc.

• But
  – We generally assume that \texttt{D is stable} (compared to \texttt{q})
    • Many, many queries addressing the same set of docs
  – We usually only search for terms (after tokenization)

• Both properties can be exploited to pre-compute a term-based \texttt{full-text index} over \texttt{D}
Inverted Files

- Very simple and effective **index structure** for terms \( K \) in a collection \( D \) of documents
- “Bag of words” approach
  - We give up on order of terms in docs (reappears later)
  - We cannot reconstruct docs based on index only
- Start from “docs contain terms” (~docs) and invert to “terms appear in docs” (=index)

<table>
<thead>
<tr>
<th>docs</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D7</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D8</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Doc1:**
Now is the time for all good men to come to the aid of their country.

**Doc2:**
It was a dark and stormy night in the country manor. The time was past midnight.
Boolean Retrieval

- We can now efficiently implement Boolean queries
- For each query term $k_i$, look-up doc-list $D_i$ containing $k_i$
- Evaluate query in the usual order
  - $k_i \land k_j : D_i \cap D_j$
  - $k_i \lor k_j : D_i \cup D_j$
  - NOT $k_i : D \backslash D_i$
- Example

\[(\text{time AND past AND the}) \ OR \ (\text{men})\]
\[= (D_{\text{time}} \cap D_{\text{past}} \cap D_{\text{the}}) \cup D_{\text{men}}\]
\[= (\{1,2\} \cap \{2\} \cap \{1,2\}) \cup \{1\}\]
\[= \{1,2\}\]
Necessary and Obvious Tricks

• Looking-up $D_i$ by scanning the entire inverted file: $O(|K|)$
• How do we efficiently look-up doc-list $D_i$?
  – Bin-search on inverted file: $O(\log(|K|))$
  – Inefficient: Essentially random access on IO, improvements soon

• Computing union/intersection naively requires $O(|D_i|^2)$
• How do we support union and intersection efficiently?
  – Keep doc-lists sorted all the time
  – Intersection $D_i \cap D_j$: Sort-Merge is $O(|D_i| + |D_j|)$
  – Union $D_i \cup D_j$: Sort-Merge is $O(|D_i| + |D_j|)$
  – If $|D_i| \ll |D_j|$, use binsearch in $D_j$ for all terms in $D_i$
    • Whenever $|D_i| + |D_j| > |D_i|*\log(|D_j|)$
Necessary and Less Obvious Tricks

- Doc-lists might get very large
  - Consumes considerable memory, danger of swapping
- How to keep size of intermediate results low?
  - Consider selectivity of each term in query
  - Obviously, selectivity $s(k_i) \sim \frac{DF_i}{|D|}$
- Only conjunctions
  - Sort terms in decreasing selectivity
  - Expected size of result is
    \[ |q| = |D| \lor sel(q) = |D| \land \prod_{i\in q} sel(k_i) \]
    - Assuming independence of terms
    - We never need more than $O(\max(|D_i|))$ memory
- General queries
  - Optimization problem: Find optimal order of evaluation
  - $sel(D_i \cap D_j) = sel(D_i) \land sel(D_j)$
  - $sel(D_i \cup D_j) = sel(D_i) \lor sel(D_j) - (sel(D_i) \land sel(D_j))$
Adding Frequency

- Implementing the VSM using TF*IDF requires storing the term frequency of all terms in all docs in corpus.
- Split up the inverted file into a dictionary and a posting list.
Size

- **Size of the dictionary**
  - Number of different terms in $D (= O(|K|)$
  - Consequence of Zipf’s law: From a certain corpus size on, new terms appear only very infrequently
    - But there are always new terms, no matter how large $D$
    - Example: 1GB text (TREC-2) generates only 5MB dictionary
      - *Typically: <1 Million*
        - More in German due to “zusammengesetzte Substantive”
  - Size of the posting list
    - Much larger: Worst case is $O(|K|*|D|)$, average case is much less

- **Implementation**
  - Dictionary should always fit into *main memory*
    - Fixed size of elements, simple array-based implementation
  - Posting list remains on disk
Storing the Dictionary

- Dictionary are always kept in main memory
  - Allows very fast answers for terms that do not appear in D
- Suitable data structure?
  - Let \(|K|\) be the number of terms in K, and \(n\) there total length
- AVL-Tree: **Balanced binary tree** build over all keywords
  - Can be build in \(O(\ n*\log(n)\ )\)
  - Searching costs \(O(\ \log(|K|)\ )\)
- Much better
  - Keyword-tree
  - Can be build in \(O(n)\), space is \(O(n)\)
  - Since we are not looking for all keywords in a doc, but only for the pointer from a keyword to the posting file, searching is \(O(|k|)\)
    - We don’t need output / failure links
Example

This is also called a trie

<table>
<thead>
<tr>
<th>Term</th>
<th>N docs</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>aid</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>all</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>come</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>country</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>dark</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>for</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>good</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>in</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>is</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>manor</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>men</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>midnight</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>night</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>now</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Posting file
Improvement: Compact Trie

- Remove all nodes with only one child
- Label edges with substrings, not single characters
- Saves pointers and jumps
  - Not too many for natural language: The tree usually is almost “full”
Storing the Posting File

- Posting file is usually kept on disk
- Thus, we need an IO-optimized data structure
- Suggestions?
- Recall DB implementation
  - Reserve large number of consecutive blocks on disk
  - Each posting has an offset
  - Reserve minimal number of bytes per posting (> one doc ID)
    - Many overflows may be handled internally
  - Upon overflow, append new posting list at the end of the file
    - Place pointer at old position – at most two access per posting
    - Again, reserve more space than immediately necessary
- Deleting docs?
  - Usually, start free-list management, but deletions are rare in IR …
**INSERT $d_{\text{new}}$**

- **What has to be done?**
  - Let $D_{\text{new}} = \{k_1, \ldots, k_m\}$, $n = |k_1| + \ldots + |k_m|$
  - Foreach $k_i$
    - Search $k_i$ in dictionary trie
    - If present
      - Follow pointer to posting file
      - Add $d_{\text{new}}$ to posting list of $k_i$
      - If list overflows, move posting list to end of file and place pointer
    - If not present
      - Insert $k_i$ after *longest common prefix* in trie
      - Add new posting list \{$d_{\text{new}}$\} at end of posting file

- **Complexity**
  - All searching is $O(n)$ (including inserting new terms) in main mem.
  - Accessing posting is at most two IO per keyword
    - Expensive, but caching helps
Building a Large Inverted File

- Doing “INSERT D_{new}” all the time is not a good idea
  - We will search the same terms all over again
- Better: Hierarchical construction
  - Reserve “sufficient” space for expected size of dictionary
    - May be estimated: Heap’s law says that a text of n words will have \( O(k^n \beta) \) distinct words, with \( k \sim 10-20 \) and \( \beta \sim 0.4-0.6 \)
  - Repeat until D is finished
    - Add docs until memory is full
      - Postings are kept in whatever order as linked lists
    - Flush postings in sorted order
  - Merge posting lists using sort-merge
    - If we have less than \(|\text{memory}|\) files, open all and merge in one run
    - Otherwise, perform hierarchical merge
    - Also consider using larger buffers for having more sequential IO
Content of this Lecture

- Inverted Files
  - Phrase and proximity search
  - Using a RDBMS
- Signature Files
  - S-Trees
Positional Information

- What if we **search for phrases**: “Bill Clinton”, “Ulf Leser”
  - ~10% of web searches are phrase queries
- What if we **search by proximity** “car AND rent/5”
  - “We rent cars”, “cars for rent”, “special care rent”, “if you want to rent a car, click here”, “Cars and motorcycles for rent”, …
- We need to add **positional information** to the posting list

---

**Doc1:**
Now is the time for all good men to come to the aid of their country.

It was a dark and stormy night in the country manor. The time was past midnight.

<table>
<thead>
<tr>
<th>Word</th>
<th>TF</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>night</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>now</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>of</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>past</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>stormy</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>the</td>
<td>1,2</td>
<td>2</td>
</tr>
<tr>
<td>their</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>time</td>
<td>1,2</td>
<td>9</td>
</tr>
<tr>
<td>to</td>
<td>1,2</td>
<td>15</td>
</tr>
<tr>
<td>was</td>
<td>1,2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
Proximity Search

• Phrase search = proximity search with distance one
• Proximity search
  - Search doc-lists with positional information for each term
  - Upon intersection, consider doc-ID and position information
  - Can get quite involved for multi-term queries
    - “car AND rent/5 AND cheap/2 AND toyota/20” – “cheap” between 1 and 7 words from “car”, “toyota” between 1 and 22 words from rent …
    - All conditions must be satisfied
  - Higher selectivity, takes about the same time as multi-term search
• Space requirements
  - We now have one number for every term in D
  - Inverted file plus positional information is typically 30-50% larger than the document collection itself
Content of this Lecture

- Inverted Files
  - Phrase and proximity search
  - Using a RDBMS

- Signature Files
  - S-Trees
Implementing an Inverted File using a RDBMS

- A rather simple model
Example Query 1

- Boolean: All docs containing terms “night” and “to”
  
  ```sql
  SELECT D1.docid
  FROM terms T1, terms T2, termdoc D1, termdoc D2
  WHERE T1.term='night' AND T2.term='to' AND
  D1.termid=T1.termid AND
  D2.termid=T2.termid AND
  D1.docid = D2.docid;
  ```

<table>
<thead>
<tr>
<th>Term-ID</th>
<th>Term</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Night</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>To</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term-ID</th>
<th>Doc-ID</th>
<th>Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term-ID</th>
<th>Doc-ID</th>
<th>TF</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example Query 2

- **VSM queries**
  - We need to compute the inner product of two vectors
    - Query and doc
  - In the query, all TF-values of query terms are 1, others are 0
    - Thus, we only need to aggregate all TF values of query terms per document
  - We ignore normalization

- **Example:** Compute score for query “night rider” (two terms)
  - `SELECT did, sum(tf)
    FROM (  SELECT D.docid did, T.term term, tf
            FROM terms T, termdoc D
            WHERE T.term='night' D.termid=T.termid) docs
    UNION
    SELECT D.docid did, T.term term, tf
    FROM terms T, termdoc D
    WHERE T.term='rider' D.termid=T.termid) docs
    WHERE docs.term in ('rider', 'night')
    GROUP BY did;`
Access Methods

- Use B*-Indices on ID columns
- Searching a term
  - Requires $O(\log(|K|))$ random-access IO
    - Mind the base of the logarithm: Block size
    - For <100M terms, this usually means at most 3 IO (top-most in cache)
  - Accessing the posting list: $O(\log(n))$ random-access IO
    - Where $n$ is the number of term occurrences in $D$
    - Access is a lookup with term-ID, then seq. can along the B*-leaves
  - Compared to IR: Dictionary in memory, posting is accessed by direct link, then only sequential IO
- Advantages: Simple, easy to build
- Disadvantages: Usually slower
  - More IO, general RDBMS overhead
  - Difficult to keep exactly the dictionary in memory
Content of this Lecture

• Inverted Files
  – Phrase and proximity search
  – Using a RDBMS

• Signature Files
  – S-Trees
Signature Files [Hen07]

• Term-based **bit-oriented hash index**

• Definition

  *The signature of length $f$ of a term $k$ is a bitstring of $f$ bits.*

• Remark

  – We obtain signatures through hashing
  
  – Exemplary hash function

```plaintext
hash := 0;
for $i := 1$ to $l$ do
    hash := $(hash + w_i) \times 157$;
end
hash := hash mod $2^F$;
```

  – We don’t need collision-free hash functions
Naïve Usage

- The simplest way of using signatures (AND-query)
  - Compute list $L$ of pairs ($docid$, signatures) for all terms in each doc
  - Given a query, convert terms into their signatures
  - Scan $L$ for each signature
  - If signature is found, check in doc whether term is really there
    - Could be a false positive because hash function is not collision-free
    - Compute intersection of doc-lists
- Advantage (little): Comparing two signatures is faster than comparing to strings
- Disadvantage: Too much IO
Superimposed Coding

- **Idea:** Superimpose signatures of all terms in a block
  - Block: Document or a fixed-size portion of the entire corpus
  - All signatures within a block are merged into one by taking their UNION (OR)

<table>
<thead>
<tr>
<th>Term</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>100010010001100</td>
</tr>
<tr>
<td>Retrieval</td>
<td>000100001010100</td>
</tr>
<tr>
<td>Information</td>
<td>001010110011000</td>
</tr>
<tr>
<td>Block signature</td>
<td>101101111011100</td>
</tr>
</tbody>
</table>

- **Search phase**
  - Compute signatures of all terms in query
  - Scan through block signatures
  - Upon match, check doc for being false positive
    - Chances for false positives are higher now than for term signatures

- **Advantages:** Much smaller index, less IO during scan
Estimating Probability of False Positives

- We assume that **blocks are fixed size** (b signatures)
- We also assume that all signatures have at most m bits set
  - The latter isn’t difficult to achieve: Use m hash functions, each returning a number between 1 and f
  - We now assume “exactly m bits”, i.e., we are overestimating the error
- Let’s look at a block signature
  - Probability of a particular bit to be set by a particular term: \( m/f \)
  - Probability of a particular bit to be set by any term in block:
    \[
    1 - \left( 1 - \frac{m}{f} \right)^b
    \]
  - Probability, that all m bits of a single term query are set:
    \[
    \left( 1 - \left( 1 - \frac{m}{f} \right)^b \right)^m
    \]
  - We assume that all matches are false ones
  - From this, we can derive an optimal m value: \( m = f \cdot \ln(2)/b \)
  - See [Hen07] for details
Example

- We can now trade space \((b, f)\) for speed (probability of FPs)
- Assume we want an index which is 10% of the size of the corpus
  - Let \(n\) be the size of D in bytes, \(k\) be the number of terms in D
  - Size of block signature file: \((k*f)/b/8\) bytes
  - Setting \(0.1*n=k*f/b/8\), we get \(f/b = 8*0.1*n/k\)
  - We chose \(m = f/b*\ln(2) = 8*0.1*n*\ln(2)/k\)
  - With an average word length \(8\) (=\(n/k\)), we get \(m\sim6\)
    - Probability of false positives is \(\sim2.14\%\)
- With a 20% index, we have a prob. of \(\sim0.04\%\)
- Thus, only 4 our of 10.000 docs will match by chance
Conclusions Signature Files

• Advantage
  – Block signatures lead to small index sizes
  – Scanning is fast due to pure bitstring comparisons

• Disadvantages
  – Requires sequential scan (of a very small index)
  – But: With larger memory sizes, signature indexes can be kept in memory
  – This makes the method attractive (now and in the future)
Content of this Lecture

• Inverted Files
  – Building and searching inverted files
  – Phrase and proximity search
  – Using a RDBMS

• Signature Files
  – S-Trees
S-Trees

- We can do better than scanning the signature file sequentially?
- Definition
  An **S-tree is a tree structured index** with
  - Each inner node contains $k$ pairs (signature, pointer to other node). For a pair $(s,p)$, $s$ must be the union of the signatures in $p$.
  - Each leaf node contains $k$ pairs (signature, docid)
Searching in S-Trees

- We assume a single-term query $k$
- Let $q$ be the signature of $k$
- We start at root and scan all signatures $s_i$
  - In an inner node, if $k = k \land s_i$, follow the pointer recursively
  - In a leaf, if $k = k \land s_i$, add doc-id to doc-list

- Usually, **multiple branches** have to be followed
- Still, large portions of the tree are pruned
Inserting into an S-Tree

• Obviously, we want to explore few branches upon searching

• This can be achieved if similar signatures are put into the same branch
  – Avoid setting to many 1

• Inserting a new signature $s$
  – Start at root and scan signatures
  – Choose signature which is the most similar to $s$
    • Similar: Hamming distance
  – Follow pointer until in leaf node and insert

• Overflow: Split node (how?), may propagate up the tree
  – Similar to $B^*$-tree