11-442 / 11-642:
Search Engines

Index Creation

Jamie Callan
Carnegie Mellon University
callan@cs.cmu.edu

Lecture Outline

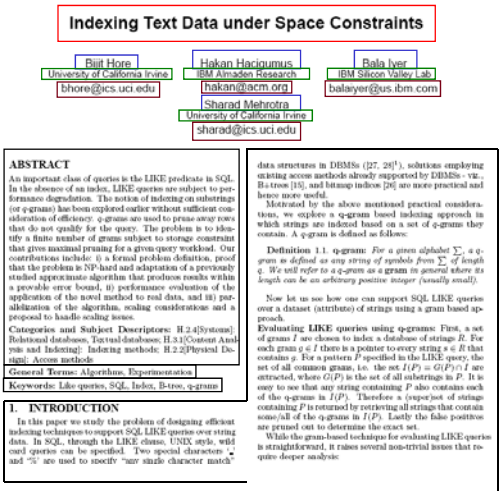
- **Building inverted lists on a single processor**
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- **Forward indexes**
- **Storing document structure**
- **Indri and Lucene indexes**
- **Index updates**

Unstructured vs. Structured Documents

We have covered indexing unstructured documents

- A single bag-of-words for the entire document

We also want to index structured documents


Indexing Text Data under Space Constraints

Bijit Hore, University of California Irvine, bhore@ics.uci.edu
Hakan Hacigumus, IBM Almaden Research, hakan@acm.org
Bala Iyer, IBM Silicon Valley Lab, balaiyer@us.ibm.com
Sharad Mehrotra, University of California Irvine, sharad@ics.uci.edu

ABSTRACT

An important class of queries is the LIKE predicate in SQL. In the absence of an index, LIKE queries are subject to performance degradation. The notion of indexing on substrings (or q-grams) has been explored earlier without sufficient consideration of efficiency. q-grams are used to prune away rows that do not qualify for the query. The problem is to identify a finite number of grams subject to storage constraint that gives maximal pruning for a given query workload. Our contributions include: i) a formal problem definition, proof that the problem is NP-hard and adaptation of a previously studied approximate algorithm that produces results within a provable error bound, ii) performance evaluation of the application of the novel method to real data, and iii) parallelization of the algorithm, scaling considerations and a proposal to handle scaling issues.

Categories and Subject Descriptors: H.2.4[Systems]: Relational databases; Textual databases; H.3.1[Content Analysis and Indexing]: Indexing methods; H.2.2[Physical Design]: Access methods

General Terms: Algorithms, Experimentation

Keywords: Like queries, SQL, Index, B-tree, q-grams

1. INTRODUCTION

In this paper we study the problem of designing efficient indexing techniques to support SQL LIKE queries over string data. In SQL, through the LIKE clause, UNIX style, wild card queries can be specified. Two special characters '_' and '%' are used to specify "any single character match"

data structures in DBMSs ([27, 28][1]), solutions employing existing access methods already supported by DBMSs - viz., B+-trees [15], and bitmap indices [26] are more practical and hence more useful.

Motivated by the above mentioned practical considerations, we explore a q-gram based indexing approach in which strings are indexed based on a set of q-grams they contain. A q-gram is defined as follows:

Definition 1.1. q-gram: *For a given alphabet $\sum$, a q-gram is defined as any string of symbols from $\sum$ of length q. We will refer to a q-gram as a* gram *in general where its length can be an arbitrary positive integer (usually small).*

Now let us see how one can support SQL LIKE queries over a dataset (attribute) of strings using a gram based approach.

Evaluating LIKE queries using q-grams: First, a set of grams I are chosen to index a database of strings R. For each gram $g \in I$ there is a pointer to every string $s \in R$ that contains g. For a pattern P specified in the LIKE query, the set of all common grams, i.e. the set $I(P) = G(P) \cap I$ are extracted, where $G(P)$ is the set of all substrings in P. It is easy to see that any string containing P also contains each of the q-grams in $I(P)$. Therefore a (super)set of strings containing P is returned by retrieving all strings that contain some/all of the q-grams in $I(P)$. Lastly the false positives are pruned out to determine the exact set.

While the gram-based technique for evaluating LIKE queries is straightforward, it raises several non-trivial issues that require deeper analysis:

49

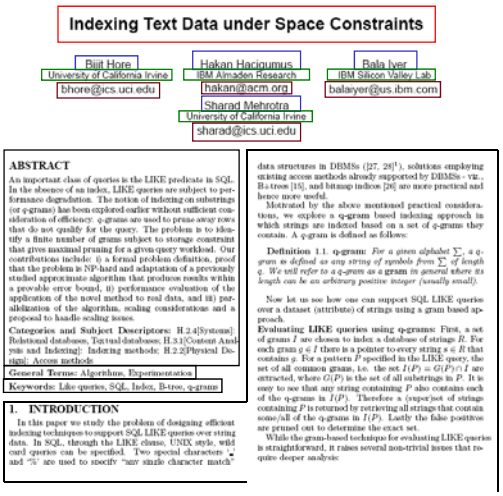
Structured Documents

The different parts of a document are called

- Fields (older terminology)
- Elements (XML terminology)

There are three sources of document structure

- Explicit markup
- Different text representations
- Annotations
  - "inferred markup"


Indexing Text Data under Space Constraints

Bijit Hore, University of California Irvine, bhore@ics.uci.edu
Hakan Hacigumus, IBM Almaden Research, hakan@acm.org
Bala Iyer, IBM Silicon Valley Lab, balaiyer@us.ibm.com
Sharad Mehrotra, University of California Irvine, sharad@ics.uci.edu

ABSTRACT

An important class of queries is the LIKE predicate in SQL. In the absence of an index, LIKE queries are subject to performance degradation. The notion of indexing on substrings (or q-grams) has been explored earlier without sufficient consideration of efficiency. q-grams are used to prune away rows that do not qualify for the query. The problem is to identify a finite number of grams subject to storage constraint that gives maximal pruning for a given query workload. Our contributions include: i) a formal problem definition, proof that the problem is NP-hard and adaptation of a previously studied approximate algorithm that produces results within a provable error bound, ii) performance evaluation of the application of the novel method to real data, and iii) parallelization of the algorithm, scaling considerations and a proposal to handle scaling issues.

Categories and Subject Descriptors: H.2.4[Systems]: Relational databases; Textual databases; H.3.1[Content Analysis and Indexing]: Indexing methods; H.2.2[Physical Design]: Access methods

General Terms: Algorithms, Experimentation

Keywords: Like queries, SQL, Index, B-tree, q-grams

1. INTRODUCTION

In this paper we study the problem of designing efficient indexing techniques to support SQL LIKE queries over string data. In SQL, through the LIKE clause, UNIX style, wild card queries can be specified. Two special characters '_' and '%' are used to specify "any single character match"

data structures in DBMSs ([27, 28][1]), solutions employing existing access methods already supported by DBMSs - viz., B+-trees [15], and bitmap indices [26] are more practical and hence more useful.

Motivated by the above mentioned practical considerations, we explore a q-gram based indexing approach in which strings are indexed based on a set of q-grams they contain. A q-gram is defined as follows:

Definition 1.1. q-gram: *For a given alphabet $\sum$, a q-gram is defined as any string of symbols from $\sum$ of length q. We will refer to a q-gram as a* gram *in general where its length can be an arbitrary positive integer (usually small).*

Now let us see how one can support SQL LIKE queries over a dataset (attribute) of strings using a gram based approach.

Evaluating LIKE queries using q-grams: First, a set of grams I are chosen to index a database of strings R. For each gram $g \in I$ there is a pointer to every string $s \in R$ that contains g. For a pattern P specified in the LIKE query, the set of all common grams, i.e. the set $I(P) = G(P) \cap I$ are extracted, where $G(P)$ is the set of all substrings in P. It is easy to see that any string containing P also contains each of the q-grams in $I(P)$. Therefore a (super)set of strings containing P is returned by retrieving all strings that contain some/all of the q-grams in $I(P)$. Lastly the false positives are pruned out to determine the exact set.

While the gram-based technique for evaluating LIKE queries is straightforward, it raises several non-trivial issues that require deeper analysis:

50

Structured Documents: Explicit Markup

<COURSE>
<TITLE> Search Engines <TITLE>
<COURSE ID> 11-442 / 11-642 </COURSE ID>
<DESCRIPTION> This course provides a comprehensive introduction to the theory and ... </DESCRIPTION>
<INSTRUCTOR> Jamie Callan </INSTRUCTOR>
<TIME> Tu/Th 10:30 - 11:50 </TIME>
<TEXT> This course studies the theory, design … </TEXT>
…
</COURSE>

Structured Documents: Multiple Representations

<DOC>
<TITLE> See New Viral Videos: Bull in Crowd </TITLE>
<BODY>
A Spanish sporting exhibition went horribly awry when a disgruntled bull leapt into the stands and began forcefully interacting with spectators. 40 onlookers were injured …
</BODY>
<URL> spike, channel, viralvideo, bull in crowd </URL>
<INLINK>
bull in crowd video, bull jumps into crowd, 40 people hurt, crazy video, Spanish bull fights back, bullfighting tragedy, ...
</INLINK>
</DOC>

Structured Documents: Named Entity (NE) Annotations

I have been looking and looking for a new camera to replace our bulky , but simple and reliable (but only fair picture taker) Sony/**ORGANIZATION** Mavica FD 73. My other choice (Besides the more expensive Nikon/**ORGANIZATION** Coolpix 3100) was the (also more expensive) Sony/**ORGANIZATION** Cybershot P 72. I recommend any of these cameras , and I was set to buy the Sony/**ORGANIZATION**, but at the last minute I cheaped out and bought the 2100/**DATE**. No regrets. I bought the camera (along with 128 mb memory card) the stock 16 mb card will be kept in the bag as a spare (and carrying case) at the new Best Buy in Harrisburg/**LOCATION** , PA/**ORGANIZATION**. I also bought a set of 4 …

Note: /O is dropped for words that aren't entities

Structured Documents

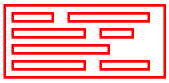
The 3 types of structure have different characteristics

- **Explicit markup**
  - High probability of being correct
  - Typically describes document organization
- **Multiple representations**
  - Different ways of representing the document meaning
- **Annotations**
  - Likely to have some amount of error
  - Sometimes describes document organization (e.g., sentence)
  - Often specifies a semantic type (e.g., person)

Indexing Structured Documents: Two Typical Approaches

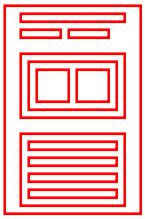
Treat each element as independent of other elements

- This makes sense for fielded data
  - E.g., why mix TITLE, DATE, AUTHOR?
  - E.g., Medical records
- **Advantage:** Simple architecture



Treat elements as part of an element hierarchy

- This makes sense for XML documents
  - E.g., DOCUMENT ⊇ SECTION ⊇ SUBSECTION
  - E.g., Scientific papers, government regulations
- **Advantage:** Flexible, may better match user needs



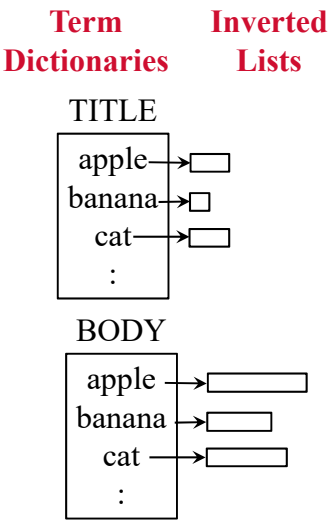
Storing Fields and Structure Using Separate Vocabularies

When elements are independent, a simple approach is to use separate vocabularies for each field

A simple implementation

- Treat query terms as a combination of FIELD and TERM information
  - E.g., TITLE::cat, BODY::cat
  - E.g., cat.TITLE, cat.BODY
  - E.g., "aspartame" [MeSH Terms] "aspartame" [All Fields]

Simple, efficient, effective for shallow structure



Storing Fields and Structure Explicitly

Complex structure requires a more sophisticated approach

- E.g., XML documents
- Terms in "Subsection" should also appear in "Section"
- Using separate vocabularies would require storing the same information repeatedly, once for each containing field

Solution: Additional data structures that store document structure

- Essentially store the parse tree…with additional information
  - E.g., the length of the element
  - E.g., pointers to its parent or children

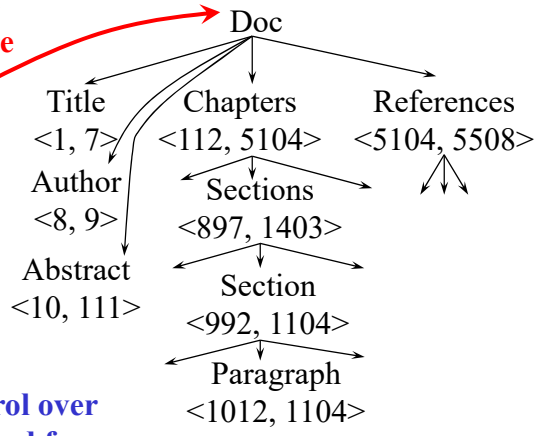
Storing Fields and Structure as Trees

Traditional inverted list

doc: 28
tf: 3
loc: 6
loc: 27
loc: 5442
doc: 92
:

Inverted list with structure

doc: 28
struct:
tf: 3
loc: 6
loc: 27
loc: 5442
doc: 92
:



This gives query-time control over which term locations are used for matching and scoring

Storing Fields and Structure as Trees

The tree-based approach is general … but where are trees stored?

- Trees can be big if documents have detailed structure
  - 20 bytes/node × 100 nodes/doc × 1,000,000 docs = 2 GB
- Significant I/O (disk) vs significant memory costs (memory)

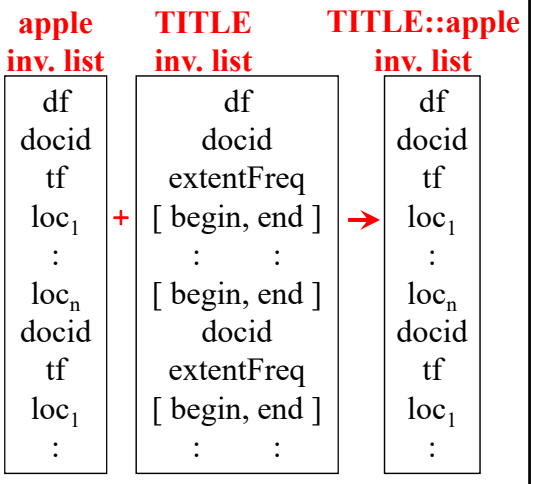
Storing Fields and Structure as Inverted Lists

Element extents can be stored as inverted lists

- Only the structure needed for this query is accessed
- Efficient for elements that aren't in every document

Slower than using separate vocabularies

- But not a lot slower
- And a lot more flexible



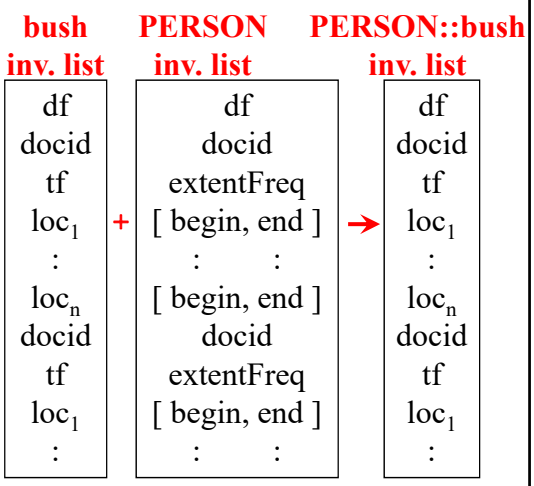
Storing Fields and Structure as Inverted Lists

Element extents can be stored as inverted lists

- Only the structure needed for this query is accessed
- Efficient for elements that aren't in every document

Slower than using separate vocabularies

- But not a lot slower
- And a lot more flexible



Lecture Outline

- Building inverted lists on a single processor
- Inverted lists and inverted files
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Storing document structure
- Indri and Lucene indexes
- Index updates

Indri Index Components

- **Manifest:** Metadata about the index
- **Statistics files**
- **Term dictionary**
- **Inverted lists**
- **Term lists (forward index)**
- **Compressed collection**

Indri Index Components: Statistics Files

Indri has three types of statistics index files

Filename	Description	Size
documentLengths	Docid→length 4 bytes/doc	101M
documentStatistics	DocumentData: length stopped length unstopped unique term count offset in directFile length of list in bytes	605M
frequentTerms	term statistics	7.2M
	Total	**0.7 GB**

gov2 corpus (450 GB of text)

(David Fisher)

Indri Index Components: Term Dictionaries

Indri has two term dictionaries

- One for frequent terms
  - Terms that occur more than 1,000 times
- One for rare terms
- Both term dictionaries use a B-Tree data structure

Filename	Description	Size	
frequentID	string → id	8.3M	**185,000 terms**
frequentString	id → string	7.9M	
infrequentID	string → id	1.6G	**39.78 million terms**
infrequentString	id → string	1.6G	
	Total	**3.2 GB**	

gov2 corpus (450 GB of text)

(David Fisher)

Indri Index Components: Inverted Files

Indri has two inverted files

- One for terms
- One for fields

Filename	Description	Size
fieldsFile	field inverted lists RVL compressed	380M
invertedFile	term inverted lists, RVL compressed	41G
	Total	**41.4 GB**

RVL: Restricted Variable Length compression

gov2 corpus (450 GB of text)

(David Fisher)

Indri Index Components: Term Lists (Forward Index)

Indri has a document term list

Filename	Description	Size
directFile	docid→term list RVL compressed	40G
	Total	**40 GB**

RVL: Restricted Variable Length compression

int terms [ ]
43121
34127
0
0
25434
9982
98476
12653
43121
34376
0
: :

gov2 corpus (450 GB of text)

(David Fisher)

Indri Index Components: The Compressed Collection

Compressed documents stored in the collections subdirectory

Filename	Description	Size
manifest	xml manifest file	4.1k
forwardLookup0	keyfile docid → metadata value	863M
reverseLookup0	keyfile metadata value → docid	454M
lookup	keyfile docid → offset in storage	459M
storage	compressed ParsedDocuments, zlib	134G
	Total	**136G**

Forward metadata elements have forwardLookup<n> keyfiles

Reverse metadata elements have reverseLookup<n> keyfiles

- E.g. above, docid (internal to external, external to internal)

(David Fisher)

Lucene Indexes: Logical Organization of the Index

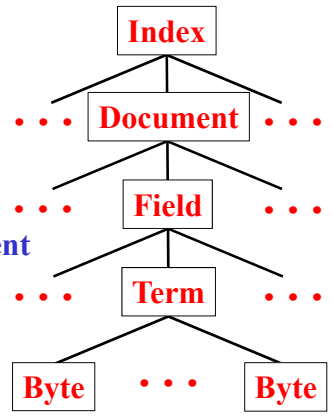
An index is a sequence of documents

- A document is a sequence of fields
- A field is a sequence of terms
- A term is a sequence of bytes

The same sequence of bytes in two different fields is considered a different term

- A term can be in multiple fields, thus is represented as [field name, bytes]
- E.g., [title, apple]

This is consistent with what we have seen so far



69

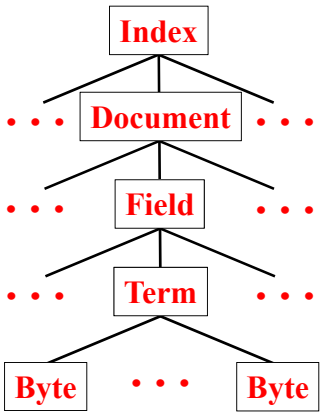
Lucene Indexes: Segments

An index can have segments

- Each segment is an independent index
- Segments can be searched independently
- Indri provides the same capability

Ids are only valid within a segment

- E.g., each segment has a docid 0



70

Lucene Indexes: Overview

Segments maintain the files shown on the right

- segmentName.fileType

A segment's files may be combined into one file

- A compound file (.cfs, .cfe)
- This reduces the number of open file handles
- E.g., the index for HW1

Type of Information	Filetypes
Segment info	.si
Field names	.fnm
Stored field values	.fdt, .fdx
Term dictionary	.tim, .tip
Term frequency data	.doc
Term proximity data	pos, .pay
Normalization factors	.nvd, .nvm
Per-document values	.dvd, .dvm
Term vectors	.tvx, .tvd, .tvf
Deleted documents	.del

Lucene Indexes: Index Segment Components

Segment info (.si): Metadata

Field names (.fnm): The names of fields used in the index

Stored field values (.fdx, .fdt):

- For each document, [attribute, value] pairs
- E.g., external docid, title, …
- Used for display purposes

Lucene Indexes: Field Indexing

Types of fields

- **Tokenized:** Contents are tokenized, the tokens are indexed
  - url field: http www cs cmu edu callan Papers
- **Not tokenized:** The contents are treated as a single token
  - url field: http://www.cs.cmu.edu/~callan/Papers/
- **Indexed:** Inverted access (inverted lists)
- **Stored:** Non-inverted access (forward index)

A field can be both stored and indexed

Lucene Indexes: Index Segment Components

Term dictionary (.tim, .tip)

- Includes df, pointer to frequency data, and pointer to proximity data

Term frequency data (.frq)

- Inverted lists for each term, containing [docid, tf]

Term proximity data (.prx):

- Inverted lists for each term, containing each position

Lucene Indexes: Index Segment Components

Normalization factors (.nrm.cfs, .nrm.cfe)

- For each field in the document, a multiplier

Term vectors (forward index) (.tvx, .tvd, .tff?)

- For each field in each document, the term text and frequency

Per-document values (.dv.cfs, .dv.cfe)

- Similar to stored values, but stored for fast access
- Used for scoring purposes

Lucene Indexes: Index Segment Components

Deleted documents (.del)

- An optional file indicating which docids are deleted

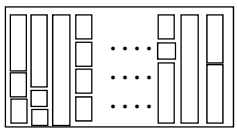
Lecture Outline

- Building inverted lists on a single processor
- Inverted lists and inverted files
  - Inverted list compression
  - Inverted list optimizations
- Forward indexes
- Storing document structure
- Indri and Lucene indexes
- Index updates

Inverted File Management: Static File (No Updates)

Access Information (Small File)

Inverted Lists (Large File)



- Create files when inverted list fragments are merged
- There is no empty space between inverted lists
- Lists are stored in canonical order (e.g., alphabetic)
- Easy to create, very space efficient
- Very difficult to update; easier to rebuild
  - Update by merging fragments with file to create new file

Updating Indexes

Indexes are expensive to update

- Suppose a new document contains 100 unique terms
- Adding that document means updating 100 inverted lists
  - Acquire lock, read list, write list, release lock
  - A lot of complexity, a lot of I/O
- Adding one document is tolerable, adding several is expensive

Updates are often done in batches

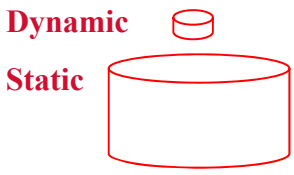
- Update every day, or after N documents arrive, or …
- Parse documents to generate index modifications
- Update each inverted list for all documents in the batch

Updating Indexes

Sometimes dynamic updates are unavoidable

- E.g., news, Twitter, …

Split index into dynamic and static parts



- The dynamic index is small
- The static index is big
- Make updates to the dynamic index
  - Acquire lock, read list, update list, write lock
  - Faster because lists are small, but still somewhat complex
- Search both static (big) and dynamic (small) components
- Periodically merge dynamic into static

Deleting Documents

Deleting a document is an expensive operation

- If the document contains N terms, must update N inverted lists
- A major problem in a system that is being used dynamically

Delete lists are a less expensive option

- When a document is deleted, add its id to a delete list
  - Don't actually delete it from the index
- When doing a search
  - Evaluate the query to produce a ranked list
  - Scan the list, removing any documents on the delete list
- When the delete list becomes large
  - Garbage collect the inverted lists, or rebuild the index

Lecture Outline

- **Building inverted lists on a single processor**
- **Inverted lists and inverted files**
  - Inverted list compression
  - Inverted list optimizations
- **Forward indexes**
- **Storing document structure**
- **Indri and Lucene indexes**
- **Index updates**

Overview of the Indexing Unit

Indexing architecture

- Document manager, parser, indexer

Inverted lists and files

- Inverted list formats
- Access mechanisms
- How they are built (single processor, multiprocessor)
- Storing document structure
- Inverted list optimizations

Term dictionary

Forward indexes

Index updates

For More Information

- I.H. Witten, A. Moffat, and T.C. Bell. "Managing Gigabytes." Morgan Kaufmann. 1999.
- G. Salton. "Automatic Text Processing." Addison-Wesley. 1989.
- J. Zobel and A. Moffat. "Inverted files for text search engines." *ACM Computing Surveys*, 38 (2). 2006.