WIKIPEDIA

Search engine indexing

Search engine indexing collects, parses, and stores data to facilitate fast and accurate information retrieval. Index design incorporates interdisciplinary concepts from linguistics, cognitive psychology, mathematics, informatics, and computer science. An alternate name for the process in the context of search engines designed to find web pages on the Internet is *web indexing*.

Popular engines focus on the full-text indexing of online, natural language documents.[1] Media types such as video[2] and audio[3] and graphics[4] are also searchable.

Meta search engines reuse the indices of other services and do not store a local index, whereas cache-based search engines permanently store the index along with the corpus. Unlike full-text indices, partial-text services restrict the depth indexed to reduce index size. Larger services typically perform indexing at a predetermined time interval due to the required time and processing costs, while agent-based search engines index in real time.

Contents

Indexing
- Index design factors
- Index data structures
- Challenges in parallelism
- Inverted indices
- Index merging
- The forward index
- Compression

Document parsing
- Challenges in natural language processing
- Tokenization
- Language recognition
- Format analysis
- Section recognition
- HTML Priority System
- Meta tag indexing

See also

References

Further reading

Indexing

The purpose of storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every document in the corpus, which would require considerable time and computing power. For example, while an index of 10,000 documents can be queried within milliseconds, a sequential scan

of every word in 10,000 large documents could take hours. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval.

Index design factors

Major factors in designing a search engine's architecture include:

Merge factors
How data enters the index, or how words or subject features are added to the index during text corpus traversal, and whether multiple indexers can work asynchronously. The indexer must first check whether it is updating old content or adding new content. Traversal typically correlates to the data collection policy. Search engine index merging is similar in concept to the SQL Merge command and other merge algorithms.[5]

Storage techniques
How to store the index data, that is, whether information should be data compressed or filtered.

Index size
How much computer storage is required to support the index.

Lookup speed
How quickly a word can be found in the Inverted index. The speed of finding an entry in a data structure, compared with how quickly it can be updated or removed, is a central focus of computer science.

Maintenance
How the index is maintained over time.[6]

Fault tolerance
How important it is for the service to be reliable. Issues include dealing with index corruption, determining whether bad data can be treated in isolation, dealing with bad hardware, partitioning, and schemes such as hash-based or composite partitioning,[7] as well as replication.

Index data structures

Search engine architectures vary in the way indexing is performed and in methods of index storage to meet the various design factors.

Suffix tree
Figuratively structured like a tree, supports linear time lookup. Built by storing the suffixes of words. The suffix tree is a type of trie. Tries support extendable hashing, which is important for search engine indexing.[8] Used for searching for patterns in DNA sequences and clustering. A major drawback is that storing a word in the tree may require space beyond that required to store the word itself.[9] An alternate representation is a suffix array, which is considered to require less virtual memory and supports data compression such as the BWT algorithm.

Inverted index
Stores a list of occurrences of each atomic search criterion,[10] typically in the form of a hash table or binary tree.[11][12]

Citation index
Stores citations or hyperlinks between documents to support citation analysis, a subject of Bibliometrics.

Ngram index
Stores sequences of length of data to support other types of retrieval or text mining.[13]

Document-term matrix

Used in latent semantic analysis, stores the occurrences of words in documents in a two-dimensional sparse matrix.

Challenges in parallelism

A major challenge in the design of search engines is the management of serial computing processes. There are many opportunities for race conditions and coherent faults. For example, a new document is added to the corpus and the index must be updated, but the index simultaneously needs to continue responding to search queries. This is a collision between two competing tasks. Consider that authors are producers of information, and a web crawler is the consumer of this information, grabbing the text and storing it in a cache (or corpus). The forward index is the consumer of the information produced by the corpus, and the inverted index is the consumer of information produced by the forward index. This is commonly referred to as a **producer-consumer model**. The indexer is the producer of searchable information and users are the consumers that need to search. The challenge is magnified when working with distributed storage and distributed processing. In an effort to scale with larger amounts of indexed information, the search engine's architecture may involve distributed computing, where the search engine consists of several machines operating in unison. This increases the possibilities for incoherency and makes it more difficult to maintain a fully synchronized, distributed, parallel architecture.[14]

Inverted indices

Many search engines incorporate an inverted index when evaluating a search query to quickly locate documents containing the words in a query and then rank these documents by relevance. Because the inverted index stores a list of the documents containing each word, the search engine can use direct access to find the documents associated with each word in the query in order to retrieve the matching documents quickly. The following is a simplified illustration of an inverted index:

Inverted Index

Word	Documents
the	Document 1, Document 3, Document 4, Document 5, Document 7
cow	Document 2, Document 3, Document 4
says	Document 5
moo	Document 7

This index can only determine whether a word exists within a particular document, since it stores no information regarding the frequency and position of the word; it is therefore considered to be a boolean index. Such an index determines which documents match a query but does not rank matched documents. In some designs the index includes additional information such as the frequency of each word in each document or the positions of a word in each document.[15] Position information enables the search algorithm to identify word proximity to support searching for phrases; frequency can be used to helto the query. Such topics are the central research focus of information retrieval.

The inverted index is a sparse matrix, since not all words are present in each document. To reduce computer storage memory requirements, it is stored differently from a two dimensional array. The index is similar to the term document matrices employed by latent semantic analysis. The inverted index can be considered a form of a hash table. In some cases the index is a form of a binary tree, which requires additional storage but may reduce the lookup time. In larger indices the architecture is typically a distributed hash table.[16]

Index merging

The inverted index is filled via a merge or rebuild. A rebuild is similar to a merge but first deletes the contents of the inverted index. The architecture may be designed to support incremental indexing,[17] where a merge identifies the document or documents to be added or updated and then parses each document into words. For technical accuracy, a merge conflates newly indexed documents, typically residing in virtual memory, with the index cache residing on one or more computer hard drives.

After parsing, the indexer adds the referenced document to the document list for the appropriate words. In a larger search engine, the process of finding each word in the inverted index (in order to report that it occurred within a document) may be too time consuming, and so this process is commonly split up into two parts, the development of a forward index and a process which sorts the contents of the forward index into the inverted index. The inverted index is so named because it is an inversion of the forward index.

The forward index

The forward index stores a list of words for each document. The following is a simplified form of the forward index:

Forward Index

Document	Words
Document 1	the,cow,says,moo
Document 2	the,cat,and,the,hat
Document 3	the,dish,ran,away,with,the,spoon

The rationale behind developing a forward index is that as documents are parsed, it is better to immediately store the words per document. The delineation enables Asynchronous system processing, which partially circumvents the inverted index update bottleneck.[18] The forward index is sorted to transform it to an inverted index. The forward index is essentially a list of pairs consisting of a document and a word, collated by the document. Converting the forward index to an inverted index is only a matter of sorting the pairs by the words. In this regard, the inverted index is a word-sorted forward index.

Compression

Generating or maintaining a large-scale search engine index represents a significant storage and processing challenge. Many search engines utilize a form of compression to reduce the size of the indices on disk.[19] Consider the following scenario for a full text, Internet search engine.

- It takes 8 bits (or 1 byte) to store a single character. Some encodings use 2 bytes per character[20][21]
- The average number of characters in any given word on a page may be estimated at 5 (Wikipedia:Size comparisons)

Given this scenario, an uncompressed index (assuming a non-conflated, simple, index) for 2 billion web pages would need to store 500 billion word entries. At 1 byte per character, or 5 bytes per word, this would require 2500 gigabytes of storage space alone. This space requirement may be even larger for a fault-tolerant distributed storage architecture. Depending on the compression technique chosen, the index can be reduced to a fraction of this size. The tradeoff is the time and processing power required to perform compression and decompression.

Notably, large scale search engine designs incorporate the cost of storage as well as the costs of electricity to power the storage. Thus compression is a measure of cost.

Document parsing

Document parsing breaks apart the components (words) of a document or other form of media for insertion into the forward and inverted indices. The words found are called *tokens*, and so, in the context of search engine indexing and natural language processing, parsing is more commonly referred to as tokenization. It is also sometimes called word boundary disambiguation, tagging, text segmentation, content analysis, text analysis, text mining, concordance generation, speech segmentation, lexing, or lexical analysis. The terms 'indexing', 'parsing', and 'tokenization' are used interchangeably in corporate slang.

Natural language processing is the subject of continuous research and technological improvement. Tokenization presents many challenges in extracting the necessary information from documents for indexing to support quality searching. Tokenization for indexing involves multiple technologies, the implementation of which are commonly kept as corporate secrets.

Challenges in natural language processing

Word Boundary Ambiguity

Native English speakers may at first consider tokenization to be a straightforward task, but this is not the case with designing a multilingual indexer. In digital form, the texts of other languages such as Chinese, Japanese or Arabic represent a greater challenge, as words are not clearly delineated by whitespace. The goal during tokenization is to identify words for which users will search. Language-specific logic is employed to properly identify the boundaries of words, which is often the rationale for designing a parser for each language supported (or for groups of languages with similar boundary markers and syntax).

Language Ambiguity

To assist with properly ranking[22] matching documents, many search engines collect additional information about each word, such as its language or lexical category (part of speech). These techniques are language-dependent, as the syntax varies among languages. Documents do not always clearly identify the language of the document or represent it accurately. In tokenizing the document, some search engines attempt to automatically identify the language of the document.

Diverse File Formats

In order to correctly identify which bytes of a document represent characters, the file format must be correctly handled. Search engines which support multiple file formats must be able to correctly open and access the document and be able to tokenize the characters of the document.

Faulty Storage

The quality of the natural language data may not always be perfect. An unspecified number of documents, particular on the Internet, do not closely obey proper file protocol. Binary characters may be mistakenly encoded into various parts of a document. Without recognition of these characters and appropriate handling, the index quality or indexer performance could degrade.

Tokenization

Unlike literate humans, computers do not understand the structure of a natural language document and cannot automatically recognize words and sentences. To a computer, a document is only a sequence of bytes. Computers do not 'know' that a space character separates words in a document. Instead, humans must program the computer to identify

what constitutes an individual or distinct word referred to as a token. Such a program is commonly called a tokenizer or parser or lexer. Many search engines, as well as other natural language processing software, incorporate specialized programs for parsing, such as YACC or Lex.

During tokenization, the parser identifies sequences of characters which represent words and other elements, such as punctuation, which are represented by numeric codes, some of which are non-printing control characters. The parser can also identify entities such as email addresses, phone numbers, and URLs. When identifying each token, several characteristics may be stored, such as the token's case (upper, lower, mixed, proper), language or encoding, lexical category (part of speech, like 'noun' or 'verb'), position, sentence number, sentence position, length, and line number.

Language recognition

If the search engine supports multiple languages, a common initial step during tokenization is to identify each document's language; many of the subsequent steps are language dependent (such as stemming and part of speech tagging). Language recognition is the process by which a computer program attempts to automatically identify, or categorize, the language of a document. Other names for language recognition include language classification, language analysis, language identification, and language tagging. Automated language recognition is the subject of ongoing research in natural language processing. Finding which language the words belongs to may involve the use of a language recognition chart.

Format analysis

If the search engine supports multiple document formats, documents must be prepared for tokenization. The challenge is that many document formats contain formatting information in addition to textual content. For example, HTML documents contain HTML tags, which specify formatting information such as new line starts, **bold** emphasis, and font size or style. If the search engine were to ignore the difference between content and 'markup', extraneous information would be included in the index, leading to poor search results. Format analysis is the identification and handling of the formatting content embedded within documents which controls the way the document is rendered on a computer screen or interpreted by a software program. Format analysis is also referred to as structure analysis, format parsing, tag stripping, format stripping, text normalization, text cleaning and text preparation. The challenge of format analysis is further complicated by the intricacies of various file formats. Certain file formats are proprietary with very little information disclosed, while others are well documented. Common, well-documented file formats that many search engines support include:

- HTML
- ASCII text files (a text document without specific computer readable formatting)
- Adobe's Portable Document Format (PDF)
- PostScript (PS)
- LaTeX
- UseNet netnews server formats
- XML and derivatives like RSS
- SGML
- Multimedia meta data formats like ID3
- Microsoft Word
- Microsoft Excel
- Microsoft PowerPoint
- IBM Lotus Notes

Options for dealing with various formats include using a publicly available commercial parsing tool that is offered by the organization which developed, maintains, or owns the format, and writing a custom parser.

Some search engines support inspection of files that are stored in a compressed or encrypted file format. When working with a compressed format, the indexer first decompresses the document; this step may result in one or more files, each of which must be indexed separately. Commonly supported compressed file formats include:

- ZIP - Zip archive file
- RAR - Roshal ARchive file
- CAB - Microsoft Windows Cabinet File
- Gzip - File compressed with gzip
- BZIP - File compressed using bzip2
- Tape ARchive (TAR), Unix archive file, not (itself) compressed
- TAR.Z, TAR.GZ or TAR.BZ2 - Unix archive files compressed with Compress, GZIP or BZIP2

Format analysis can involve quality improvement methods to avoid including 'bad information' in the index. Content can manipulate the formatting information to include additional content. Examples of abusing document formatting for spamdexing:

- Including hundreds or thousands of words in a section which is hidden from view on the computer screen, but visible to the indexer, by use of formatting (e.g. hidden "div" tag in HTML, which may incorporate the use of CSS or JavaScript to do so).
- Setting the foreground font color of words to the same as the background color, making words hidden on the computer screen to a person viewing the document, but not hidden to the indexer.

Section recognition

Some search engines incorporate section recognition, the identification of major parts of a document, prior to tokenization. Not all the documents in a corpus read like a well-written book, divided into organized chapters and pages. Many documents on the web, such as newsletters and corporate reports, contain erroneous content and side-sections which do not contain primary material (that which the document is about). For example, this article displays a side menu with links to other web pages. Some file formats, like HTML or PDF, allow for content to be displayed in columns. Even though the content is displayed, or rendered, in different areas of the view, the raw markup content may store this information sequentially. Words that appear sequentially in the raw source content are indexed sequentially, even though these sentences and paragraphs are rendered in different parts of the computer screen. If search engines index this content as if it were normal content, the quality of the index and search quality may be degraded due to the mixed content and improper word proximity. Two primary problems are noted:

- Content in different sections is treated as related in the index, when in reality it is not
- Organizational 'side bar' content is included in the index, but the side bar content does not contribute to the meaning of the document, and the index is filled with a poor representation of its documents.

Section analysis may require the search engine to implement the rendering logic of each document, essentially an abstract representation of the actual document, and then index the representation instead. For example, some content on the Internet is rendered via JavaScript. If the search engine does not render the page and evaluate the JavaScript within the page, it would not 'see' this content in the same way and would index the document incorrectly. Given that some search engines do not bother with rendering issues, many web page designers avoid displaying content via JavaScript or use the Noscript tag to ensure that the web page is indexed properly. At the same time, this fact can also be exploited to cause the search engine indexer to 'see' different content than the viewer.

HTML Priority System

Indexing often has to recognize the HTML tags to organize priority. Indexing low priority to high margin to labels like *strong* and *link* to optimize the order of priority if those labels are at the beginning of the text could not prove to be relevant. Some indexers like Google and Bing ensure that the search engine does not take the large texts as relevant source due to strong type system compatibility.[23]

Meta tag indexing

Specific documents often contain embedded meta information such as author, keywords, description, and language. For HTML pages, the meta tag contains keywords which are also included in the index. Earlier Internet search engine technology would only index the keywords in the meta tags for the forward index; the full document would not be parsed. At that time full-text indexing was not as well established, nor was computer hardware able to support such technology. The design of the HTML markup language initially included support for meta tags for the very purpose of being properly and easily indexed, without requiring tokenization.[24]

As the Internet grew through the 1990s, many brick-and-mortar corporations went 'online' and established corporate websites. The keywords used to describe webpages (many of which were corporate-oriented webpages similar to product brochures) changed from descriptive to marketing-oriented keywords designed to drive sales by placing the webpage high in the search results for specific search queries. The fact that these keywords were subjectively specified was leading to spamdexing, which drove many search engines to adopt full-text indexing technologies in the 1990s. Search engine designers and companies could only place so many 'marketing keywords' into the content of a webpage before draining it of all interesting and useful information. Given that conflict of interest with the business goal of designing user-oriented websites which were 'sticky', the customer lifetime value equation was changed to incorporate more useful content into the website in hopes of retaining the visitor. In this sense, full-text indexing was more objective and increased the quality of search engine results, as it was one more step away from subjective control of search engine result placement, which in turn furthered research of full-text indexing technologies.

In desktop search, many solutions incorporate meta tags to provide a way for authors to further customize how the search engine will index content from various files that is not evident from the file content. Desktop search is more under the control of the user, while Internet search engines must focus more on the full text index.

See also

- Controlled vocabulary
- Database index
- Full text search
- Information extraction
- Key Word in Context
- Selection-based search
- Site map
- Text retrieval
- Information literacy

References

1. Clarke, C., Cormack, G.: Dynamic Inverted Indexes for a Distributed Full-Text Retrieval System. TechRep MT-95-01, University of Waterloo, February 1995.