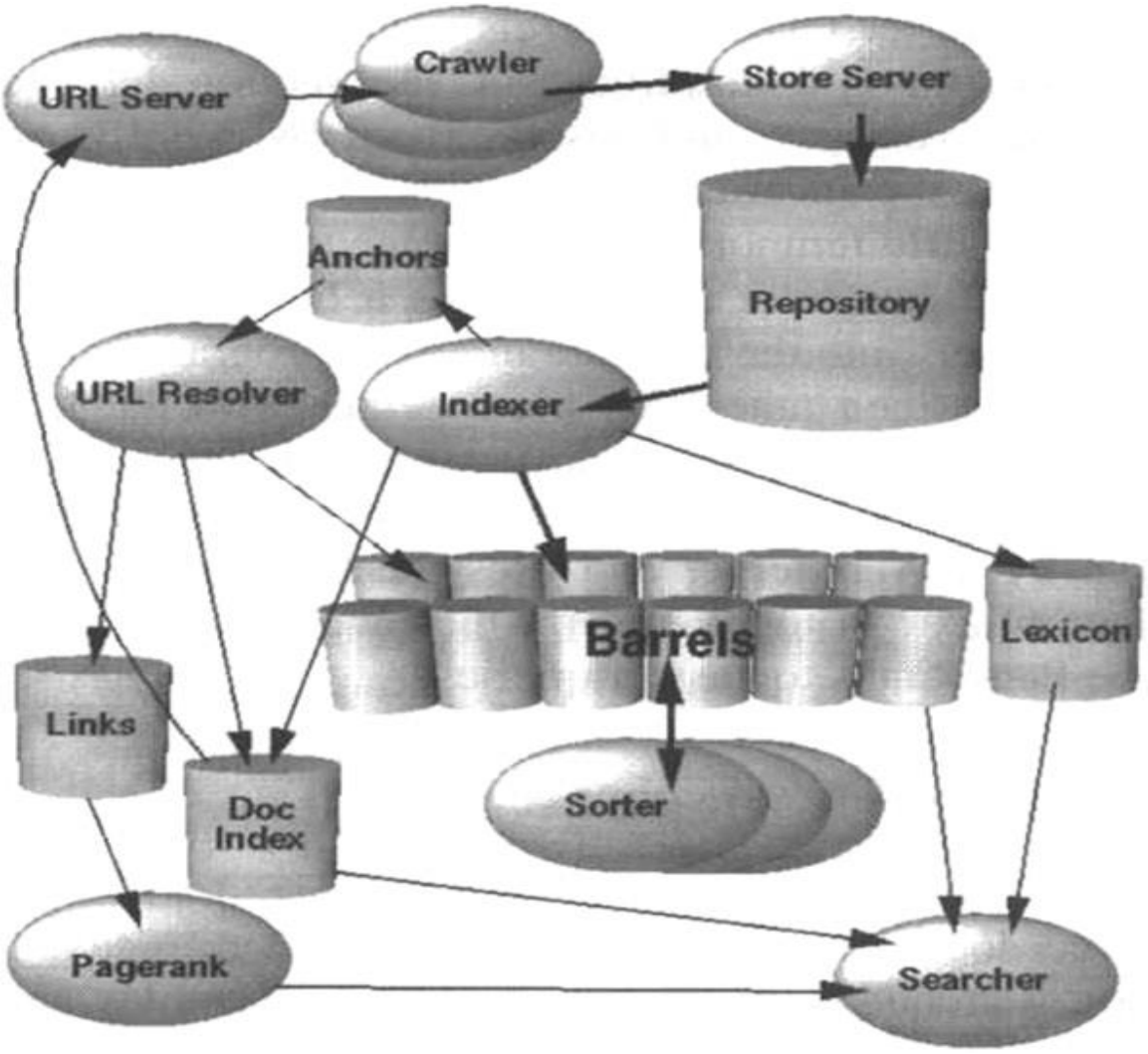
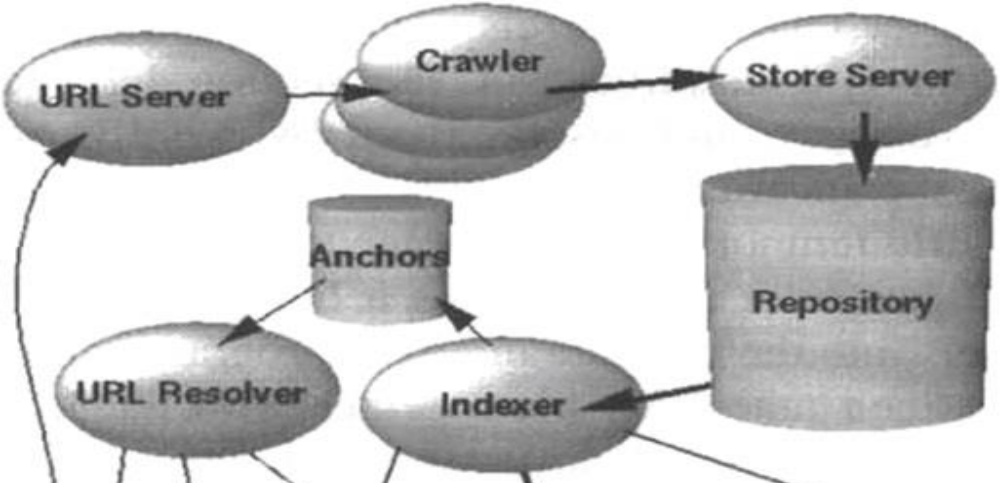
LINK ANALYSIS- GOOGLE

Module 5

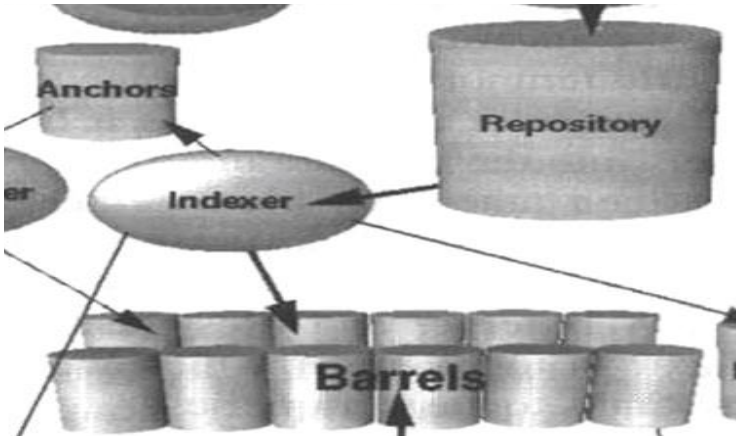
Architecture



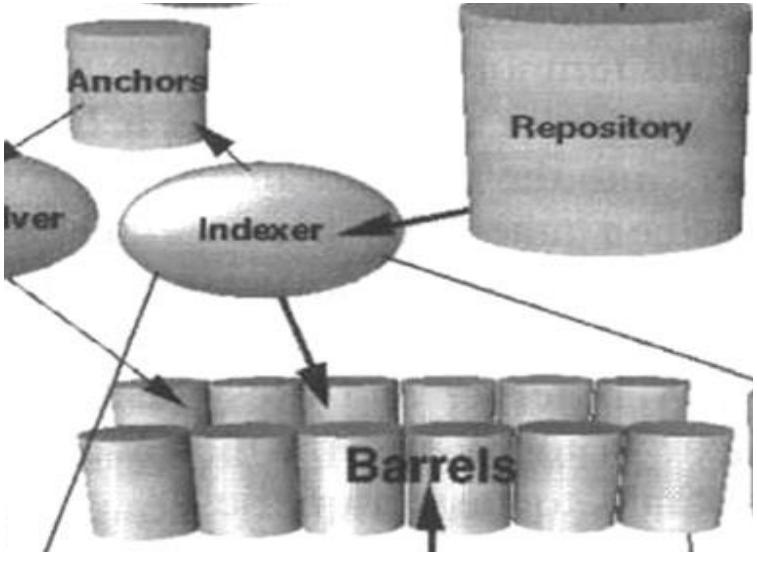
- The ***URL Server*** sends the list of URLs to be fetched by the crawlers.
- The ***Store Server*** receives the fetched pages and compresses them before storing it in the repository.
- Every Webpage has an associated ID number called a ***docID*** which is assigned whenever a new URL is parsed out of a Webpage.



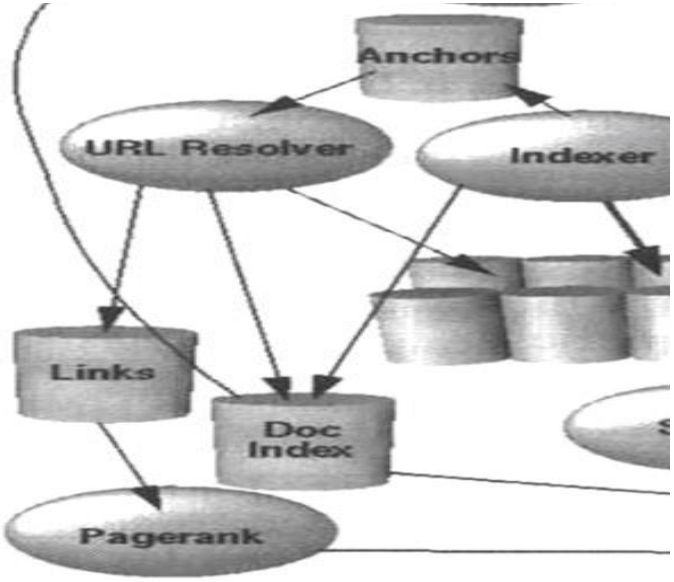
- The I***ndexer*** module reads the repository, uncompresses the documents and parses them.
- Each of these documents is converted to a set of word occurrences called ***hits***.
- The hit records the word, position in document, approximation of the font size and capitalization.
- These hits are then distributed into a set of ***Barrels***, creating a partially sorted forward index.



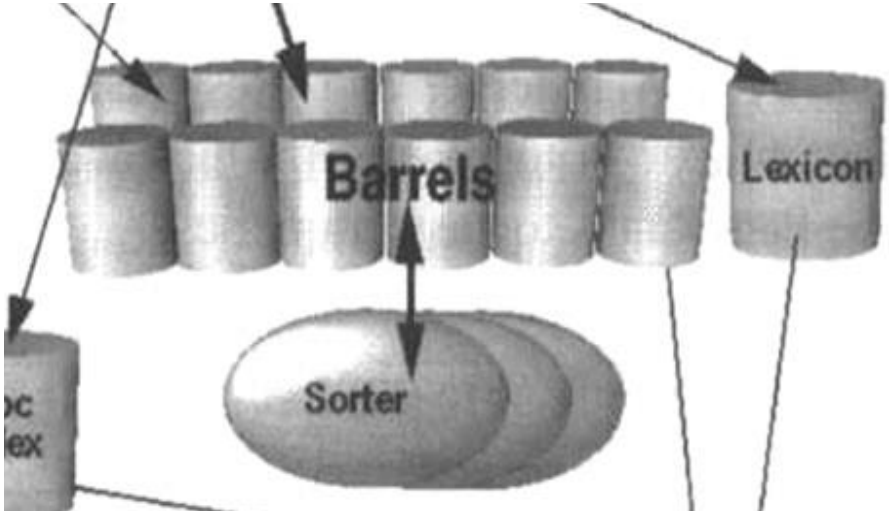
- The ***indexer*** also parses out all the links in every Webpage and stores important information about them in anchor files placed in ***Anchors***.
- These anchor files contain enough information to determine where each link points from and to, and the text of the link.



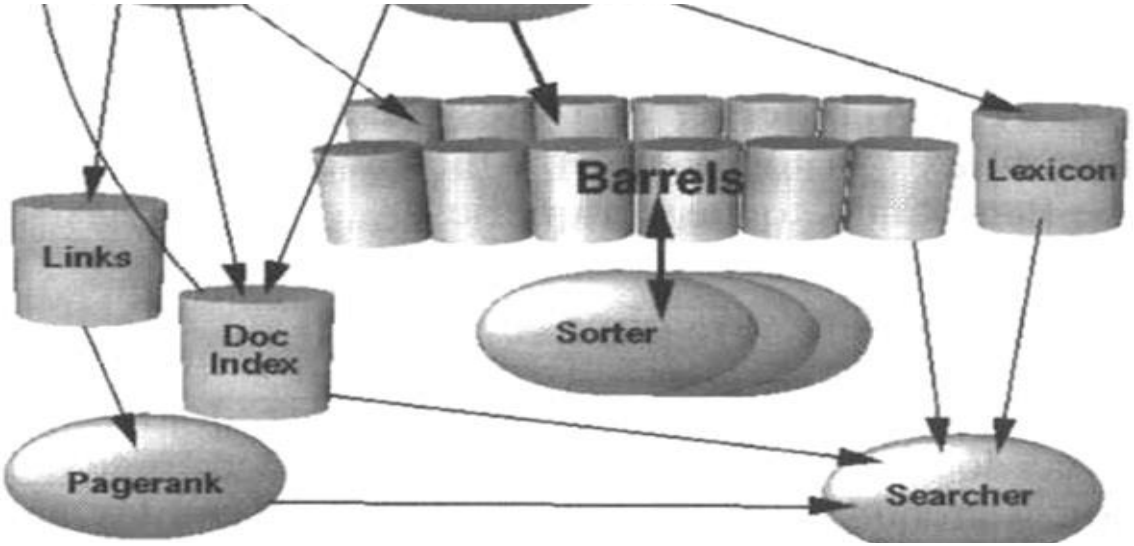
- The ***URL Resolver*** reads these anchor files and converts relative URLs into absolute URLs and in turn into docIDs.
- These anchor texts are put into the forward index, associated with the docID that the anchor points to.
- It also generates a database of links which are pairs of docIDs.
- The Links database is used to compute PageRanks for all the documents.



- The ***Sorter*** takes the barrels, which are sorted by docID, and resorts them by wordID to generate the inverted index.
- The sorter also produces a list of wordIDs and offsets into the inverted index.



- A program called ***DumpLexicon*** takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher.
- The ***Searcher*** is run by a Web server and uses the lexicon built by DumpLexicon together with the inverted index and the *PageRanks* to answer queries



Crawling

- Crawling involves interacting with hundreds of thousands of Web servers and various name servers which are all beyond the control of the system.
- In order to scale to hundreds of millions of Webpages, Google has a fast distributed crawling system.
- A single URL server serves lists of URLs to a number of crawlers.
- Both the URL server and the crawlers were implemented in Python.

- Each crawler keeps roughly 300 connections open at once.
- This is necessary to retrieve Web pages at a fast enough pace.
- A major performance stress is DNS lookup so each crawler maintains a DNS cache.
- Each of the hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request, and receiving response.
- These factors make the crawler a complex component of the system.
- It uses asynchronous IO to manage events, and a number of queues to move page fetches from state to state.

Searching

- Every hit list includes position, font, and capitalization information.
- Hits from anchor text and the PageRank of the document are factored in.
- Combining all of this information into a rank is difficult.
- For every matching document we compute counts of hits of different types at different proximity levels.
- These counts are then run through a series of lookup tables and eventually are transformed into a rank.
- This process involves many tunable parameters.

Data Structures

- Almost all of the data is stored in BigFiles which are virtual files that can span multiple file systems and support compression.
- HTML repository consists of concatenation of the compressed HTML of every page, preceded by a small header.
- The DocIndex keeps information about each document.
- The DocIndex is a fixed width Index Sequential Access Mode (ISAM) index, ordered by docID.

- The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics.
- Variable width information such as URL and title is kept in a separate file.
- There is also an auxiliary index to convert URLs into docIDs.
- The lexicon has several different forms for different operations.
- They all are memory-based hash tables with varying values attached to each word.

- Hit lists account used in both the forward and the inverted indices.
- Because of this, it is important to represent them as efficiently as possible.
- Therefore, a hand optimized compact encoding is used which uses two bytes for every hit.
- To save space, the length of the hit list is stored before the hits and is combined with the wordID in the forward index and the docID in the inverted index.

- The forward index is stored in a number of barrels.
- Each barrel holds a range of wordIDs.
- If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordIDs with hitlists which correspond to those words.
- This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter.

- The inverted index consists of the same barrels as the forward index.
- Except that they have been processed by the sorter.
- For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into.
- It points to a list of docIDs together with their corresponding hit lists.
- This list is called a doclist and represents all the occurrences of that word in all documents.

- An important issue is in what order the docIDs should appear in the doclist.
- One simple solution is to store them sorted by docID.
- This allows for quick merging of different doclists for multiple word queries.
- Another option is to store them sorted by a ranking of the occurrence of the word in each document.
- This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start.
- However, merging is much more difficult.

- Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index.
- A compromise between these options was chosen, keeping two sets of inverted barrels - one set for hit lists which include title or anchor hits and another set for all hit lists. This way, we check the first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

Web Spam Pages

- Web spam pages are notorious for using techniques to achieve higher-than-deserved rankings in a search engine's results.
- A technique to semi automatically separate reputable pages from spam.
- First, a small set of seed pages are evaluated to be good by a human expert.
- Once these reputable seed pages have been identified, the link structure of the Web is used to discover other pages that are likely to be good.

- The algorithm first selects a small seed set of pages whose spam status needs to be determined by a human.
- Using this, the algorithm identifies other pages that are likely to be good based on their connectivity with the good seed pages.
- The human checking of a page for spam is formalized as a ***oracle function O*** over all pages p ∈ V

$$O(p) = \begin{cases} 0 \; if \;\; p \; is \; bad \\ 1 \; if \;\; p \; is \; good \end{cases}$$

- Oracle evaluations over all pages is an expensive ordeal, this is avoided by asking the human to assign oracle values for just a few of the pages.
- To evaluate pages without relying on O, the likelihood that a given page p is good is estimated using a ***trust function*** **T** which yields the probability that a page is good,
- i.e, **T (p) = P[O(p) = 1].**
- By introducing a threshold value δ, we define the ***threshold trust property*** as
- **T (p) > δ ⇔ O(p) = 1**

- ***Trust functions***
- Given a budget *L* of *O*- invocations, select at random a seed set *S* of *L* pages and call the oracle on its elements.
- The subset of good and bad seed pages are denoted as *S*+ and *S*−, respectively.
- Since the remaining pages are not checked by human expert, these are assigned a trust score of 1⁄2 to signal our lack of information.
- Therefore, this scheme is called the ***ignorant trust function*** T_0 defined for any $p \in V$ as

$$T_0(p) = \begin{cases} O(p) \; if \; p \in \mathcal{S} \\ 1/2 \; otherwise \end{cases}$$

- Then compute the trust scores, by taking advantage of the approximate isolation of good pages.
- Select at random the set S of L pages that we invoke the oracle on.
- Then, expecting that good pages point to other good pages only, we assign a score of 1 to all pages that are reachable from a page in S+ in M or fewer steps.
- The ***M-step trust function*** is defined as

$$T_M(p) = \begin{cases} O(p) \; if \; p \in \mathcal{S} \\ 1 \; if \; p \notin \mathcal{S} \; and \; \exists q \in \mathcal{S}^+ : q \rightsquigarrow_M p \\ 1/2 \; otherwise \end{cases}$$

- There are two possible schemes to reduce trust from the good seed pages.
- The first is called the *trust dampening*.
- If a page is one link away from a good seed page, it is assigned a dampened trust score of β.
- A page that can reach another page with score β, gets a dampened score of $\beta \cdot \beta$.

- The second technique is called *trust splitting*.
- Here the trust gets split if it propagates to other pages.
- If a page *p* has a trust score of *T (p)* and it points to *ω(p)* pages, each of the *ω(p)* pages will receive a score fraction *T(p)/ω(p)* from *p*.
- In this case, the actual score of a page will be the sum of the score fractions received through its in-links.

- The next task is to identify pages that are desirable for the seed set.
- By desirable means that, pages that will be the most useful in identifying additional good pages.
- Ensure that the size of the seed set is reasonably small to limit the number of oracle invocations.
- There are two strategies for accomplishing this task.

- **1.** it is based on the idea that since trust flows out of the good seed pages, give preference to pages from which we can reach many other pages.
- Seed set can be built from those pages that point to many pages that in turn point to many pages and so on.
- This approach is the inverse of the PageRank algorithm because the PageRank ranks pages based on its in-degree while here it is based in the out-degree.
- This gives the technique the name *inverted PageRank*.
- While it does not guarantee maximum coverage, its execution time is polynomial in the number of pages.

- **2.** Take pages with high PageRank as the seed set, since high-PageRank pages are likely to point to other high- PageRank pages.
- Piecing all of these elements together gives us the *Trust Rank* algorithm.
- The algorithm takes as input the graph.
- At the first step, it identifies the seed set.
- The pages in the set are re-ordered in decreasing order of their desirability score.
- Then, the oracle function is invoked on the *L* most desirable seed pages.
- The entries of the static score distribution vector *d* that correspond to good seed pages are set to 1.
- After normalizing the vector *d* , the Trust Rank scores are evaluated.

- A spam farm is a group of interconnected nodes involved in link spamming.
- It has a single *target node*, whose ranking the spammer intends to boost by creating the whole structure.
- A farm also contains *boosting nodes*, controlled by the spammer and connected so that they would influence the PageRank of the target.
- Boosting nodes are owned either by the author of the target, or by some other spammer.
- Boosting nodes have little value; they only exist to improve the ranking of the target.

- A term called *spam mass* is introduced which is a measure of how much PageRank a page accumulates through being linked to by spam pages.
- The idea is that the target pages of spam farms, whose PageRank is boosted by many spam pages, are expected to have a large spam mass.
- At the same time, popular reputable pages, which have high PageRank because other reputable pages point to them, have a small spam mass.

- The *absolute spam mass* of a node x, denoted by M_x , is the PageRank contribution that x receives from spam nodes.
- Therefore, the spam mass is a measure of how much direct or indirect in-neighbour spam nodes increase the PageRank of a node.
- The *relative spam mass* of node x, denoted by mx , is the fraction of x's PageRank due to contributing spam nodes,