- Comment on Assignment (Guy)
- Goals of HotBot
- Internal Architecture
  1. Semantics
  2. Hardware
  3. Data Base Model
  4. Score
  5. Filters
- Optimizations
  1. Caching
  2. Compression
  3. Parallelism
  4. Misc.
- Crawling
- Summary
- References

1 Comment on Assignment (Guy)

Problem 3

Problem 3 in the last assignment could be done by combining the affine gap and space efficiency examples given in class and in the readings. There, however, were a few caveats which needed to be addressed.

First, the algorithm must keep separate pointers for the E and S entries. This is necessary because the path could have come from an E or S crossing. The F entries can be generated on the fly.

Second, gaps which cross the middle row must be dealt with carefully (Fig. 152). If not the algorithm could double charge for that gap. A solution is to store both pointers back to the middle row for both E and S.

Only four arrays of size O(m) are required in a space efficient algorithm (see Fig. 153). They hold E, S, and the pointers pointing back to the location in the middle row from where E and S came.

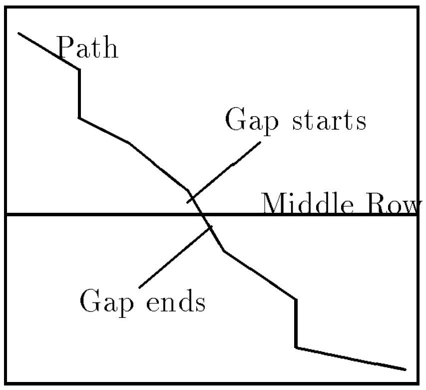

Figure 152: A gap may cross the middle row

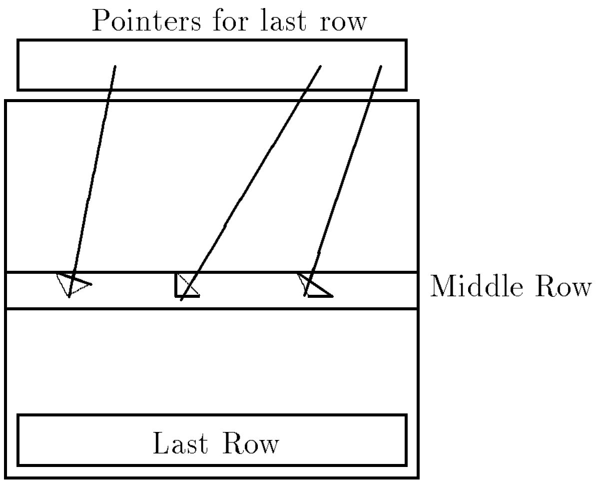

Figure 153: Keep two arrays for each of S and E

2 Goals of HotBot

History

HotBot originated in the Berkeley NOW project. An application was needed that would show the benefits of clustering. A search engine was decided upon as the killer app. because its requirements closely matched what cluster computing could provide.

Availability

In order to achieve 24x7 uptime, a mission critical infrastructure is necessary. Fault tolerance and redundancy are cornerstones in achieving this goal. The multiple individual machines in clusters provide a natural method.

Scalability

Additionally, a search engine must expand its service as client requests grow. Since growth in the WEB, however, is occurring at a staggering rate, big, expensive machines, quickly would become overloaded, with replacement the only upgrade option. So, incremental scalability could provide a large economic advantage over the longer term.

Not only the hardware, but the software design must take all this into account. Fine grained parallelism and multi-processor support should adapt well in this environment.

Both the availability and scalability requirements for a search engine could be addressed by the NOW clusters.

3 Internal Architecture

3.1 Semantics

The semantics of a search engine are simple. Although the search engine itself must be highly available, that data it contains does not. The WEB can always be used as a last recourse to retrieve any requested information.

Additionally, a very weak consistency model can be maintained. Known as BASE [1], it allows stale information to persist for a limited time.

3.2 Hardware

Currently 330 UltraSPARC CPUs power HotBot. The system is very homogeneous, which allows a node to take over for an arbitrary one when necessary. These are connected to a vast number of RAID disk subsystems which contain over 0.5TB of data.

If a single node goes down, the data associated with that node becomes temporarily unavailable. This is okay due to the weak consistency model. Also, this constitutes only a small fraction of the total data. Note that the system as a whole, however, continues to process requests despite this.

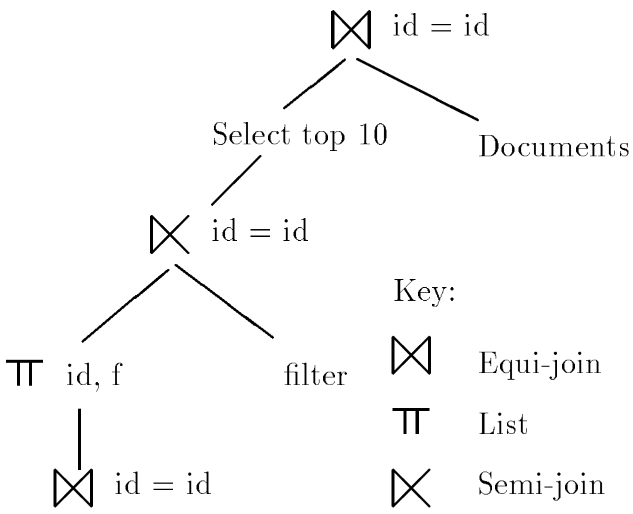

Figure 154: An example of a search query

In comparison to a single, large, powerful SMP system, HotBot's commodity parts are faster and cost less. For example, in order to expand the service, additional, ordinary systems can be bought, adding their power to the current set. SMP's, however, are not additive in this respect, nor do they provide any form of fault tolerance in system up-time. At the very least, replication is necessary, requiring additional, expensive hardware.

3.3 Data Base Model

The search engine algorithms can be described in relational data base (RDBMS) terms. There exists a relation w (a row in a table) for each unique word in each document found in the WEB: w[document ID, score, position information]. The entry keeps track of the document in which the word was found (using a unique ID). The score helps in matching the word to a search and is discussed below. The position information helps in phrase searching. There are currently about 10M words in HotBot's database with an average of about 1K lines per word. HotBot has to deal with some words which occur in almost every document and with many words that occur very infrequently.

There is also a relation D for all documents on the WEB where: D[ID, URL, etc ...]. This identifies each document with a unique ID, as well as information to retrieve the document later.

When the RDBMS needs to be updated with fresh data, an atomic swap operation swaps a new data base to replace the old between queries. This allows hot updates to occur without any interruption of service.

When a search query is received by HotBot, it goes through a series of relational joins. For example, if the query "A B filter:C" is received (i.e. search for A and B, then apply a boolean filter on the output based on C), Fig. 154 shows the relational operations. The first equi-join (lower left) is doing the search for A (think of it as an "and" operation). It then creates a list of the result, sorted by score (f). The semi-join then executes the boolean filter on the list. The top ten scores are selected and a final equi-join is done to get the associated

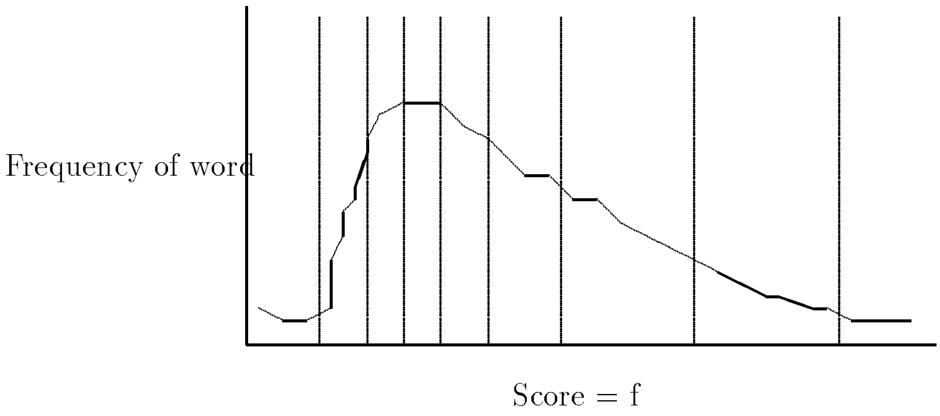

Figure 155: Determining the score

documents for the result page to return to the client.

3.4 Score

The score (f) determines when a word matches a query and how closely. Higher scores (weights) cause the document to appear higher on HotBot's priority lists to the RDBMS joins and on the result page returned to the client. The value is calculated as a function of a few different attributes, which includes:

1. Number of times a word appears in the document
2. Document length
3. If the word appears in the document title

This function is then broken down into equal area slices, and a "score" from 0 to 255 is assigned to each slice. See Fig. 155 for an example of how the function and slices could appear.

3.5 Filters

As shown in an example above, filters allow the query to express boolean matches. In fact, filters are implemented as meta-words, where w contains all documents that would match the specific filter. As a result, a filter can be thought of as the addition of this meta-word on the search line. Filters are use-full in expressing searches on features of pages that cannot be expressed otherwise (such as does the page contain an image). Note, however, that less-than and greater-than operators cannot be easily addressed by this mechanism. Dates, however, are important and do require some type of less-than, greater-than functionality. HotBot uses a trick which expands a date query into log(n) boolean queries, where "n" is the number of bits in the date specification.

4 Optimizations

In order to achieve better throughput of query transactions, optimizations help to streamline the search process.

4.1 Caching

Instead of just caching documents, HotBot also caches search results. In fact, the "next 10" button is actually another full search that completes quickly because of the cache.

4.2 Compression

Extensive compression helps to minimize the footprint of the data. Score and position information are split with IDs stored in both lists. Compression then allows the position information, which is used infrequently, to occupy minimal space. A type of delta-encoding is used (similar to Huffman) that uses a variable number of bits.

In HotBot, because of the large amount of data, nothing is uncompressed, even when loaded directly into memory, until necessary. The reasoning is that the additional page faults that would be generated by uncompressing when loading into memory outweighs the time to uncompress the data more often [2, 3].

4.3 Data Base

HotBot uses a technique called join selectivity to speed its RDBMS operations. It orders operations which have the highest probability of resulting in a short list first. This reduces the work of future RDBMS operations. For example, applying very specific boolean filters first could reduce list sizes for later equi-joins.

In addition, HotBot spreads the RDBMS data across many nodes in the cluster (by document ID in the D relation) at a relatively fine grain. Single node failures become manageable and the amount of data affected minimal. Also, when updating, the fine grains allow atomic swaps of the data to be simple and require little additional space.

4.4 Parallelism

When optimizing for parallelism, "balance" is the key word. If all nodes are working to their fullest, and there are no single bottlenecks, this allows for the highest degree of parallelism. HotBot attempts to address this by:

1. Randomizing data across the nodes

2. Use a large number of disks to hide seek latency

The randomizing reduces the risk of hot spots forming within the cluster (similar to the arguments in [4]).

HotBot also executes each query in parallel. Because each node is responsible for part of the total database, it first issues the query to each node (Fig. 156). The nodes then send the

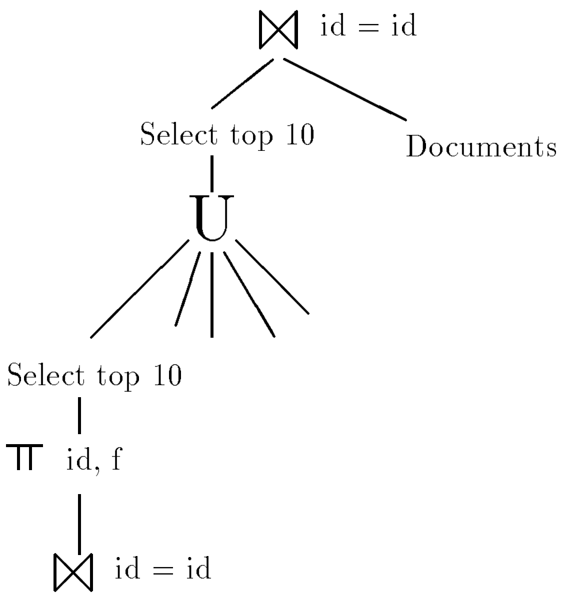

Figure 156: Parallelization of queries

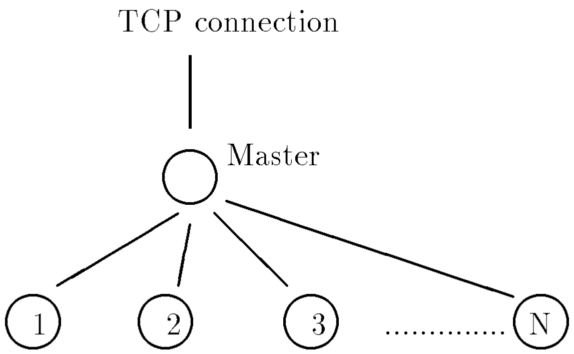

Figure 157: Naive fanout

highest scoring subset of their results back to a "master" node. This then takes the union of the answers, returns the top scores among all of the nodes responding, then does the final equi-join to associate the answers with the documents. Note that every query utilizes every node.

Another problem is how the fanout from the original master node, that initially receives the query, happens as the queries are being distributed (Fig. 157). A naive solution would be to send the queries in sequence starting from the first node. This, however, could lead to hot spots or head of line blocking (queries are artificially blocked on a resource due to a previous query) occurring in the fanout chain.

HotBot attempts to avoid problems by using two techniques (Fig. 158):

1. Fanout in multiple levels (a fat tree structure)

2. At each stage, fanout to random nodes

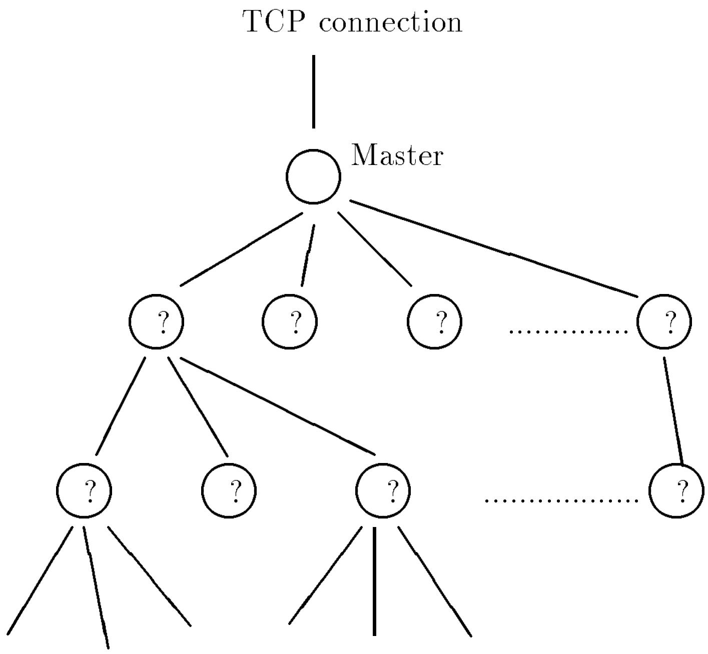

Figure 158: Optimized fanout (each ? is a random node)

In the future, HotBot could take advantage of knowledge of the physical network layout. A more optimized fanout may know where the limiting bisection bandwidths between cut-sets of the network are located and incur less traffic over those connections.

4.5 Misc.

Some additional optimizations are done to speed up each query:

1. Precomputation: As much information as possible is computed off-line. For example, the scores can be computed and stored as the pages are read in from the WEB. Also, filters (i.e. meta-words) can be created and managed before queries requesting the information arrive.
2. Partitions of the nodes are updated at varying times. This gives a better illusion that the data is up-to-date. Some data, such as news groups are updated more frequently.
3. HotBot will dynamically skip words that are too expensive to compute. This usually occurs with words that appear in many documents, such as "a" or "the".

5 Crawling

HotBot has a separate partition that crawls the WEB at about 10M pages a day. It uses "popular" sites (such as possibly Yahoo) as the roots of these searches. These updates eventually enter the HotBot database at various times.

6 Summary

The benefits gained from clustering matched the expectations placed on search engines well. Their availability, scalability, and price/performance ratio cannot be achieved even by more expensive SMP servers.

References

[1] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, *Cluster-Based Scalable Network Services*, (To appear at SOSP-16, Oct. 1997).

[2] Fred Douglis. *The Compression Cache: Using On-line Compression to Extend Physical Memory.* USENIX Proceedings, Winter 1993.

[3] D. Banks, M Stemm. *Investigating Virtual Memory Compression on Portable Architectures.* `http://HTTP.CS.Berkeley.EDU/~stemm/classes/cs262.ps.gz`

[4] Waldspurger and Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management.* Proceedings of the First Symposium of Operating System Design and Implementation, November 1994.