Efficient Query Subscription Processing for Prospective Search Engines

Utku Irmak * Svilen Mihaylov † Torsten Suel *
Samrat Ganguly ‡ Rauf Izmailov ‡

ABSTRACT

Current web search engines are retrospective in that they limit users to searches against already existing pages. Prospective search engines, on the other hand, allow users to upload queries that will be applied to newly discovered pages in the future. We study and compare algorithms for efficiently matching large numbers of simple keyword queries against a stream of newly discovered pages.

Categories and Subject Descriptors:
H.3.3 Information Storage and Retrieval: Information Search and Retrieval: Information filtering

General Terms:
Algorithms, Performance, Experimentation

Keywords:
Inverted index, prospective search, query processing

1. INTRODUCTION

The emergence of large web search engines has fundamentally changed the way we locate and access information. Such search engines work by downloading pages from the web and then building a full-text index on the pages. Thus, they are *retrospective* in nature, as they allow us to only search for currently already existing pages – including many outdated pages. An alternative approach, often called *prospective search*, allows a user to upload a query that will then be evaluated by the search engine against documents encountered in the future. The user can be notified of new matches in one of several ways, e.g., via email, or through a desktop-based or web-based RSS reader. Popular currently available implementations of prospective search include the *News Alert* feature in Google News and the PubSub subscription service (`http://pubsub.com`).

Prospective search can be performed with the help of RSS feeds. RSS (RSS 2.0: *Really Simple Syndication*) is an XML-based data format that allows web sites and weblogs to syndicate their content by making all new content, or at least meta data about new content, available at a specified location. Thus, a prospective search engine can find new content by periodically downloading the appropriate RSS feed.

In this paper, we study techniques for optimizing the performance of prospective search engines. We focus on keyword queries with particular emphasis on AND queries. However, Boolean keyword queries can also be supported easily: We convert them to DNF, insert each conjunction as a separate AND query, and remove duplicate matches from the output. For more details about this work, see [4].

*CIS Department, Polytechnic University, Brooklyn, NY 11201. {uirmak@cis.poly.edu, suel@poly.edu}. The third author was also partially supported by NSF Awards IDM-0205647 and CCR-0093400, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.
†CIS Department, University of Pennsylvania, Philadelphia, PA 19104. svilen@seas.upenn.edu.
‡NEC Laboratories America, Inc., Princeton, NJ 08540. {samrat@nec-labs.com, rauf@nec-labs.com}.

WWW 2006, May 22–26, 2006, Edinburgh, Scotland.
ACM 1-59593-332-9/06/0005.

2. THE QUERY PROCESSOR

In a naive implementation one might simply execute all the queries periodically against any newly arrived documents. However, if the number of queries is very large, this would result either in a significant delay in identifying new matches or a significant query processing load for the engine.

Each query in our system has a unique integer query ID (QID) and each term has an integer term ID (TID). Each incoming document has a unique document ID (DID) and consists of a set of TIDs, and the output of the matching algorithm consists of a stream of (QID, DID) pairs, one for each time a document satisfies a particular query. The terms in each query are ordered by TID; thus we can refer to the first, second, etc. term in a query. Any incoming documents have already been preprocessed by parsing out all terms, translating them into TIDs, and discarding any duplicate terms or terms that do not occur in any query.

The main data structure used in all our algorithms is an *inverted index*, which is also used by retrospective search engines. In our case we index the queries rather than the documents, as proposed in [5]. The inverted index consists of inverted lists, one for each unique term that occurs in the queries. Each list contains one posting for each query in which the corresponding word occurs, where a posting consists of the QID and the position of the term in the query (recall that terms are ordered within each query by TID). Since *(QID, position)* can be stored in a single 32-bit integer, each inverted list is a simple integer array.

2.1 A Primitive Matching Algorithm

The primitive matching algorithm (with some variations) has been studied in a number of works including [3, 5, 2]. The basic idea is as follows. We initially build the inverted index from the queries, and reserve space for a hash table indexed by QIDs. Given an incoming document, we clear the hash table, and then process the terms in the document in some sequential order. To process a term, we traverse the corresponding inverted list in the index. For each posting of the form (QID, position) in this list, we check if there is already an entry in the hash table for this QID. If not, we insert an entry into the table, with an associated accumulator

(counter) set to 1. If an entry already exists, we increase its accumulator by 1. This phase is called the *matching phase*. In the *testing phase*, we iterate over the hash table entries: For every entry, we test if the final value of the accumulator is equal to the number of query terms; if so we output the match between this query and the document.

2.2 Optimizations over Primitive Algorithm

Exploiting Position Information and Term Frequencies: We note that the primitive algorithm creates an entry in the hash table for any query that contains at least one of the terms. This results in a larger hash table that in turn slows down the algorithm, due to additional work that is performed but also due to resulting cache misses during hash lookups. To decrease the size of the hash table, we first exploit the fact that we are focusing on AND queries. Recall that each index posting contains (QID, position) pairs. Suppose we process the terms in the incoming document in sorted order, from lowest to highest TID. This means that for a posting with non-zero position, either there already exists a hash entry for the query, or the document does not contain any of the query terms with lower TID, and thus the query does not match. So we create a hash entry whenever the position in the posting is zero, and only update existing hash entries otherwise. A further reduction is achieved by simply assigning TIDs to terms in order of frequency (we assign TID 0 to the least frequent term). So an accumulator is only created for those queries where the incoming document contains the least frequent term in the query.

Bloom Filters: As a result of previous optimizations, hash entries are only created initially, and most of the time is spent afterwards on lookups to check for existing entries. Moreover, most of these checks are negative. To speed them up, we propose to use a Bloom filter [1], which is a probabilistic space-efficient method for testing set membership.

We use a Bloom filter in addition to the hash table. In the matching phase, when hash entries are created, we also set the corresponding bits in the Bloom filter; the overhead for this is fairly low. In the testing phase, we first perform a lookup into the Bloom filter to see if there might be a hash entry for the current QID. If the answer is negative, we immediately continue with the next posting; otherwise, we perform a hash table lookup. Since the Bloom filter structure is small, this results in fewer processor cache misses.

Partitioning the Queries: We note that the hash table and Bloom filter sizes increase linearly with the number of query subscriptions, and thus eventually grow beyond the L1 or L2 cache sizes. Instead of creating a single index, we propose to partition the queries into p subsets and build an index on each subset. An incoming document is then processed by performing the matching sequentially with each of the index partitions. While this does not decrease the number of postings traversed, or the locality for index accesses, it means that hash table and Bloom filter sizes are decreased by a factor of p, assuming we clear them after each partition.

Clustering: Our next idea is to exploit similarities between different subscriptions. In a preprocessing step, we cluster all queries into artificial *superqueries* of up to a certain size, such that every query shares the same least frequent term with a superquery. We employ greedy algorithms to construct superqueries. In the algorithms we consider queries in arbitrary order (random), in sorted order (alphabetical), or check all the candidates to maximize an overlap ratio (overlap). In related work, Fabret et al. [3] study how to cluster subscriptions for improved throughput; however the focus is on more structured queries rather than keywords.

3. EXPERIMENTAL EVALUATION

Since we were unable to find any large publicly available query subscription logs, we decided to use Excite search engine query logs, collected in 1999. We preprocessed the trace by removing stop words, and any duplicate queries, and also converting all the terms to lower case. The Excite trace contained 1077958 queries with 271167 unique terms; the resulting inverted index had 3633970 postings. To be used as incoming documents, we selected 10000 web pages at random from a large crawl of over 120 million pages from Fall 2001. To experiment with numbers of queries beyond the size of the query log, we replicated the queries according to a *multiplier* between 1 and 14 – for experiments without clustering only. We note that the first three optimizitions do not exploit similarities between different queries, and thus we believe this scaling approach is justified. The experiments are performed on a machine with a 3.0 Ghz Pentium4 processor with 16 KB L1 and 2 MB L2 cache, under a Linux 2.6.12-Gentoo-r10 environment. We used the `gcc` compiler with Pentium4 optimizations. Figure 3.1 shows the total running times of the optimizations without clustering (top), and with clustering (bottom) for matching different numbers of queries against 10000 incoming documents. Our results show that millions of subscriptions can be matched against hundreds or thousands of incoming documents per second.

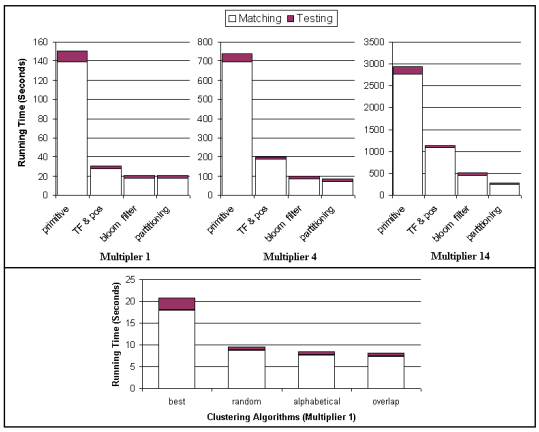

Figure 3.1: Running times of the various algorithm optimizations for different numbers of queries.

4. REFERENCES

[1] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. of ACM SIGCOMM Conf.*, 2003.

[3] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of ACM SIGMOD Conf.*, 2001.

[4] U. Irmak, S. Mihaylov, T. Suel, S. Ganguly, and R. Izmailov. Efficient Query Subscription Processing for Prospective Search Engines. In *Proc. of USENIX Annual Technical Conf.*, 2006.

[5] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems*, 24(4):529–565, 1999.