Efficiently Encoding Term Co-occurrences in Inverted Indexes

Marcus Fontoura*
Google Inc.
Mountain View, CA, USA
marcusf@google.com

Maxim Gurevich
Yahoo! Research
Santa Clara, CA, USA
maximg@yahoo-inc.com

Vanja Josifovski
Yahoo! Research
Santa Clara, CA, USA
vanjaj@yahoo-inc.com

Sergei Vassilvitskii
Yahoo! Research
New York, NY, USA
sergei@yahoo-inc.com

ABSTRACT

Precomputation of common term co-occurrences has been successfully applied to improve query performance in large scale search engines based on inverted indexes. The results of such precomputations are traditionally stored as additional posting lists in the index. During query evaluation, these precomputed lists are used to reduce the number of query terms, as the results for multiple terms can be accessed through a single precomputed list. In this paper, we expand this paradigm by considering an alternative method for storing term co-occurrences in inverted indexes. For a selected set of terms in the index, we store *bitmaps* that encode term co-occurrences. A bitmap of size k for term t augments each posting to store the co-occurrences of t with k other terms, across every document in the index. At query evaluation, size k bitmaps can be used to answer queries that involve any of the 2^k combinations of the additional terms. In contrast, a precomputed list, although typically shorter, can only be used to evaluate queries containing all of its terms. We evaluate the bitmaps technique we propose, and the baseline of adding precomputed posting lists and show that they are complementary, as they capture different aspects of the query evaluation cost. We perform an experimental evaluation on the TREC WT10g corpus and show that a hybrid strategy combining both methods significantly lowers the cost of query evaluation compared to each method separately.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Selection process

*The work was done while the author was at Yahoo! Research

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

General Terms

Algorithms, Measurement

Keywords

Inverted index, term co-occurrence, precomputation, bitmaps

1. INTRODUCTION

Inverted indexes have been successfully deployed to solve scalable retrieval problems where documents are represented as bags of terms. Each term t is associated with a *posting list*, which encodes the documents that contain t. The cost of query evaluation depends both on the number of query terms and the length of their posting lists. To reduce this cost, previous approaches [16, 18] have used precomputation of common subqueries.

This precomputation is performed in advance, during index construction, and it is then used during query evaluation where multiple query terms are replaced with one composite term. The precomputed combinations are determined based on their frequencies in logs of past usage of the system. Precomputed lists are stored in the index as regular posting lists, and thus their number is limited by the available memory. For example, terms *New* and *York* can be substituted with the precomputed list for *New York*. While this approach has been shown to be successful in reducing the query evaluation time, it is limited by the fact that each additional posting list can only be used for queries containing *all* of the pre-joined terms. Moreover, given additional available memory, only a limited number of term combinations can be precomputed. We use this precomputation technique as a baseline.

In this paper we introduce a more flexible way to encode term co-occurrence using *bitmaps*. A bitmap is associated with each posting and encodes co-occurrence of the posting term with a fixed set of other terms, chosen offline at index construction time. Given k bits we can encode the co-occurrence of k different terms with the posting list term in a list with bitmaps. This allows us to use the posting list of a single term to resolve queries involving that term and any of the 2^k combinations of the chosen terms. Had we chosen to represent each of these combinations by a separate posting list, the memory cost, as well as the complexity of picking the right combinations during query evaluation, would have become prohibitive. As they represent many

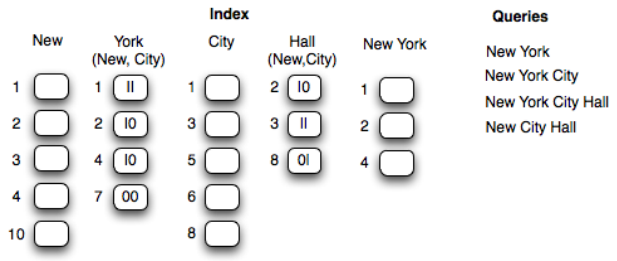

Figure 1: An example index with bitmaps for terms York and Hall and a precomputed list for New York (left) and example query workload (right).

term combinations, bitmaps are triggered more frequently during query evaluation, although this benefit is somewhat offset by the fact that postings lists with bitmaps are typically longer than precomputed lists.

In Figure 1 we present an example to illustrate the intuition behind the use of bitmaps. On the left of the figure we have an index with five posting lists corresponding to the terms *New, York, City, Hall* and the precomputed list for *New York*. Each posting list has document identifiers *(docids)* in sorted order. Besides *docids*, we represent bitmaps inside the oval. In this example each bitmap is 2 bits long. The terms corresponding to each bit in the bitmap are identified inside the parenthesis. An empty oval denotes no bitmaps for that term. There are two lists with bitmaps in this figure: York and Hall. The postings for term York contain bits for its co-occurrence with terms New and City. If we examine the bitmap for the posting of York in *docid* 2 we can see that it indicates that document 2 also contains term New but it does not contain term City. This is also apparent by the presence of *docid* 2 in New's posting list and by its absence in City's posting list.

On the right side of Figure 1 we show a sample query workload using the terms in the index. The index bitmaps have been optimized for this workload. We now provide some intuition on how to chose the terms to be represented in the bitmaps. Examining the workload we notice that the terms New and York co-occur in many of the queries. Thus, it would be beneficial to place a bit for New in York's list or vice versa. Since the posting list of York is shorter, we chose to place a bit for New in York's list. As terms City and Hall are both present in queries involving terms New and York, we can also add a bit for these two. In this example we are limited to bitmaps of size 2, we chose to add a bit for City in York's list. Now the first two queries can be evaluated by accessing only York's posting list. Similar reasoning is used to store the bits for New and City into Hall's posting list.

To contrast bitmaps with precomputed lists, consider again the query New York. This query can be answered either with the precomputed posting list New York or with York's list, which contains a bitmap that encodes co-occurrences of term New. In both cases only one list is accessed. The precomputed list has the advantage of being shorter, as it only contains the documents in the intersection of the lists for New and York. Therefore, using the precomputed list is more efficient as it contains only the *docids* that belong to the query results. On the other hand, bitmaps are more space efficient and more flexible – York's posting list can be used to answer queries York, New York, York City and New York City, while the precomputed posting list for New York can only be used in two of these cases. In this paper, we show that the bitmaps and precomputed posting lists capture *different* aspects of the query evaluation cost. These complementary properties lead to improved performance when using both techniques simultaneously.

1.1 Contributions

To determine which terms should be listed in the bitmaps of each posting list, we develop a model of the query evaluation cost as a function of the number of query terms and the lengths of the posting lists corresponding to these query terms. This cost model allows us to formulate the problem of selecting bitmap terms for each posting list as an optimization problem. Given a query workload (e.g., from a historical query log) and a memory budget, the goal is to select the optimal set of bitmap terms for all posting lists in the index, so as to minimize the evaluation cost of the workload. The optimization problem is NP-hard, but we show that the cost function is submodular, allowing for efficient approximation [15].

In this study we focus on the analysis of Boolean evaluation algorithm for conjunctive (AND) queries, which are prevalent in display advertising scenarios [24]. Boolean AND queries are also common in the first phase selection for web search queries, where the first phase results are reranked by machine learning scoring methods. We experimentally evaluate precomputed lists, bitmaps and their combination on the TREC WT10g corpus and a query workload derived from the AOL query log [20]. We show that the two techniques are complementary, as their combination achieves higher latency reduction than each technique individually. Latency reduction ranges from 25% for 3% growth in index size, to 71% for 4-fold index size increase. We also show that both precomputation techniques benefit not only past queries, contained in the workload used for building the index, but also new, previously unseen "long tail" queries.

In summary, the main contributions of this paper are the following.

- We introduce the concept of posting bitmaps as a flexible way to index term co-occurrences.
- We formally define the problem of selecting terms to precompute given a query workload and a memory budget, and propose an efficient solution for it.
- We show that bitmaps and precomputed lists complement each other, and that the combination significantly outperforms each technique individually.
- We present experimental results over the TREC WT10g corpus demonstrating the benefits of the approach in practice.

The rest of the paper is organized as follows. Section 2 contains background on indexing and query evaluation. In Section 3 we model the cost function of query evaluation and discuss the trade off between index size and evaluation time. Section 4 describes the index construction algorithms for bitmaps and precomputed lists, while Section 5 presents the corresponding query evaluation algorithms. Section 6 describes our experimental results. Section 7 reviews the related work. Finally, Section 8 presents our conclusions and future research directions.

2. PRELIMINARIES

2.1 Inverted indexes

Most search engines and information retrieval systems use inverted indexes as their main data structure for full-text indexing [28]. We remark that many modern applications are geared for high-throughput and low-latency scenarios (see, for example, [9, 24]), and consequently, the index data structures reside in main memory and are not swapped in from disk. There is a considerable body of literature on efficient index construction (e.g. [3, 6, 12, 14, 19, 26, 28]) and query evaluation (e.g. [7, 17, 26, 28]) algorithms.

In inverted indexes, the occurrence of a term t within a document d is called a *posting*. A posting has the form $\langle$*docid, payload* $\rangle$, where *docid* is the document identifier of d and where the payload is used to store arbitrary information about each occurrence of t within d. In this paper, we use part of the payload to store the co-occurrence bitmaps. The set of postings associated with a term t is stored in a *posting list*.

Each posting list is sorted in increasing order of *docid*. Often, B-trees [13] or skip lists are used to index the posting lists [12, 19]. This facilitates searching for a particular *docid* within a posting list, or for the smallest *docid* in the list greater than a given *docid*. In this work we use the following basic operations on posting lists:

1. *first():* returns the list's first posting;
2. *next():* returns the next posting or signals the end of list;
3. *search(d):* returns the first posting with *docid* $\geq d$, or end of list if no such posting exists. This operation is typically implemented efficiently using the posting lists indexes.

2.2 Query evaluation strategies

Although both precomputed lists and bitmaps can be used with different query evaluation algorithms, in this paper, we assume conjunctive Boolean queries and the document-at-a-time query evaluation model (DAAT) [23], commonly used in Web search engines. With Boolean queries no extra information, such as term weights, is required in the bitmaps and precomputed lists. In DAAT, the documents that satisfy the query are usually obtained via a join of the posting lists of the query terms. Given a conjunctive query $q = t_1 t_2 \dots t_n$, a search algorithm returns R – the set of *docids* of all documents that match all terms $t_1 t_2 \dots t_n$.

Let $L_1, L_2, \dots, L_n$ be the posting lists of terms $t_1, t_2, \dots, t_n$ respectively. A naive algorithm would scan $L_1, L_2, \dots, L_n$ and return the set of documents that appear in all the lists. Clearly, the naive algorithm accesses $\sum_{i=1}^{n} |L_i|$ postings, where $|L_i|$ is the number of postings in L_i.

A more efficient algorithm is Max Successor [8], shown in Algorithm 1. It sorts terms in ascending order of their list lengths and traverses them in parallel. In each iteration, the algorithm checks whether the current candidate document from the shortest list appears in other lists. Instead of scanning over possibly numerous postings in longer lists, this step is implemented by skipping to the first position containing *docid* greater than or equal to the current candidate. If all lists contain the current candidate, it is added to the result set and the position in the shortest list is advanced by 1. Otherwise, the algorithms advances the shortest list to the following potentially matching document. The algorithm iterates until it reaches the end of one of the lists. The advantage of this algorithm that the number of list accesses is proportional to the length of the shortest list. We use this algorithm as the basis for our analysis and evaluation.

Algorithm 1 Max Successor: $search(q)$

```
1: Assume that |L_1| ≤ |L_2| ≤ ... ≤ |L_n|
2: R ← ∅
3: d_1 ← L_1.first()
4: while d_1 is defined do
5:   for i ← 2 to n do
6:     d_i ← L_i.search(d_1)
7:     if d_1 ≠ d_i then
8:       d_1 ← max(d_i, L_1.next())
9:       break
10:    else if i = n then
11:      R ← R ⋃ d_1
12:      d_1 ← L_1.next()
13: return R
```

3. EVALUATION TIME VS. INDEX SIZE

Our work trades off the size of the index with the query evaluation speed. Generally, query latency is directly correlated with the number of postings that are accessed during the evaluation of the query. We begin by analyzing the cost of query evaluation as a function of the lengths of the postings lists that are accessed and define an analytic cost function. We then discuss the trade off between the size of the index and query evaluation performance and show that bitmaps and precomputed lists capture different aspects of this trade off.

3.1 The cost function

The latency of the Max Successor query evaluation algorithm [8] increases both when the algorithm has to access more lists, and when the lists accessed are themselves longer. Understanding the exact interplay between these two parameters is integral to making principled decisions as to which precomputed lists and bitmaps should be added to the index.

In each iteration of the algorithm the cursor in the shortest list advances by at least 1, and therefore the main loop of the algorithm (line 4) is executed at most $|L_1|$ times. For the remaining lists, in each iteration the algorithm advances the cursors to the next relevant *docid*, skipping other postings along the way (line 6). These skips are significant, as the total number of accesses to a secondary list is *sublinear* in the length of the list.

With this in mind, we considered the family of functions expressed as:

$$F(q) \;=\; |L_1| \sum_{i=2}^{n} G(|L_i|),$$

where G is the sublinear function quantifying the number of accesses to the secondary lists. We validated this cost function experimentally using the TREC WT10g corpus and the AOL query log [20], measuring the lengths of the accessed postings lists and the evaluation time for each query. Typical implementation of skipping by B-trees or binary search

suggests that the function G is logarithmic in shape. To test this hypothesis we considered $G(x)$ to be coming from the parametric family:

$$G(x) = C_1 + C_2 \log(x).$$

The best fit to the data was achieved by setting $C_1 = 12$ and $C_2 = 1$, which yielded an R^2 (R squared correlation coefficient) value of 0.65. Thus, this simple function explains almost $^2/_3$ of the variation that occurs in the data. For the remainder of the paper, we then fix the cost function for evaluating query q to be:

$$F(q) = |L_1| \sum_{i=2}^{n} (12 + \log |L_i|) .$$

3.2 Optimizing the cost function

We now describe how precomputed lists and bitmaps minimize the two components of the above cost function (1) the shortest list length $|L_1|$ and (2) the random access cost $12 + \log |L_i|$. Suppose terms t_1 and t_2 frequently occur as a subquery and assume $|L_1| \leq |L_2|$.

Using precomputed lists we would store the co-occurrences of $t_1 t_2$ as a new term t_{12}. The size of t_{12}'s list is exactly $|L_1 \cap L_2|$ as it contains only the documents in the intersection of these lists. Therefore, precomputed lists reduce not only the number of posting lists accessed during query evaluation, but also the size of these lists, potentially impacting the length of the shortest list in the query. This technique decreases both components of the cost function while increasing the index by $|L_1 \cap L_2|$ postings.

In the case of bitmaps, we add a bit to the payload of each posting in L_1, where the value of the bit is 1 for postings whose document contains t_2 and 0 otherwise. This allows the query evaluation algorithm to avoid accessing L_2, cutting the second component of the cost function, $12 + \log |L_2|$. The extra space is a bit for each posting in L_1.

To demonstrate the complementary nature of the two precomputation techniques we analyze their applicability to different queries in our evaluation dataset (described in Section 6). For each query of at least two terms we computed the *minimum relative intersection size* (MRIS) — the relative size of the shortest list resulting from an intersection of two query terms to the shortest list of a single term: $\frac{\min_{i,j} |L_i \cap L_j|}{|L_1|}$. MRIS captures the potential benefit of adding the optimal precomputed list of two terms for this particular query: the lower the MRIS, the lower evaluation cost can be achieved; and, moreover, the lower is the storage cost of that precomputed list. Figure 2 shows the cumulative distribution function of queries as a function of their MRIS. While the majority of the queries have low MRIS and thus can benefit from precomputed lists, a non-negligible fraction of queries have relatively high MRIS, where precomputed lists are less beneficial. Indeed, in our experimental results we show that the potential benefit of precomputed lists declines sharply with increasing MRIS, while the benefit of bitmaps rises moderately (see Figure 5).

We note here that adding bitmaps to postings does not prevent delta compression of the *docids*. Traditionally, for disk based indexes *docids* are compressed in compact array that is separated from the payload to pack as many *docids* as feasible in the available memory. Accessing the payload is done only when needed and requires additional I/Os. For in-memory indexing, the access penalty is a lot smaller and both options are applicable: storing the bitmap with the compressed *docids*, as well as using a separate payload array to store the bitmap. In either option the added cost of accessing the payload is captured by the C_1 and C_2 constants.

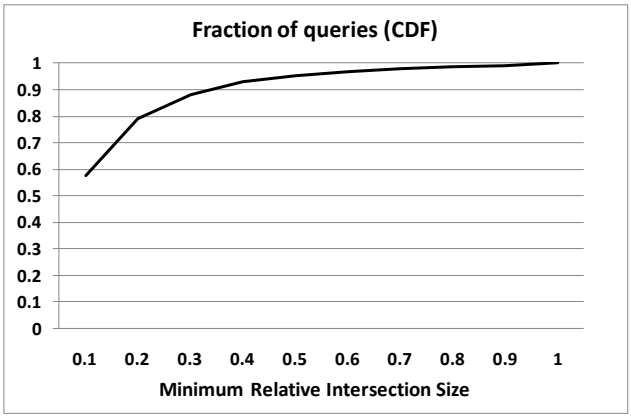

Figure 2: CDF of fraction of queries as a function of their minimum relative intersection size.

Given a typical query workload and additional memory budget, our goal is to find the optimal set of bitmaps and precomputed lists that minimizes the query evaluation cost function based on the given workload.

4. INDEX CONSTRUCTION

At index construction time, the central question is which bitmaps and precomputed lists should be added to the index to reduce the expected latency under the given query load. In both of these approaches we face a tension between the improvement in latency caused by using precomputed objects (whether bitmaps or precomputed lists) and the extra space required for precomputed results. Moreover, the problem does not decompose nicely, the contribution of a particular precomputation (whether bitmap or precomputed list) depends on the set of bitmaps and precomputed lists already added to the index. We first consider the problems of selecting bitmaps and precomputed lists separately, and then propose a hybrid algorithm for selecting them simultaneously.

4.1 Bitmaps

In the case of bitmaps the extra space required for adding a bitmap for term t_j to term t_i's list is exactly $|L_i|$ since every posting in L_i grows by one bit. The benefit of the extra bitmap varies. Suppose, for example, that we have terms New, York, and City, with list lengths $|L_{\text{New}}| \geq |L_{\text{City}}| \geq |L_{\text{York}}|$, and the queries are New York, City York, and New York City. Consider the benefit of adding a bitmap for term New to City's posting list. In the case that no previous bitmaps exist, this bitmap improves the evaluation of query New York City from $|L_{\text{York}}|(G(|L_{\text{New}}|) + G(|L_{\text{City}}|))$ to $|L_{\text{York}}|G(|L_{\text{City}}|)$. However, if the list York already has bits for terms New and City, the extra bitmap provides no additional benefit. In this case the total latency would be $|L_{\text{York}}|$ regardless of whether a bit for term New was added to City's list.

This leads us to formulate the bitmap selection problem as follows. Let B be the association matrix where $b_{ij} = 1$ if

there is a bit for term t_j in list L_i's bitmap, and 0 otherwise. For example, we set $b_{\text{City New}} = 1$ in the example above. Given a set of bitmaps B and a query q, $F(B, q)$ is the latency of evaluating q with the bitmaps indicated by B. Here we assume q is evaluated using the optimal set of lists and bitmaps minimizing the evaluation cost. Finally, let S denote the total space available for storing extra information and $Q = \{q_1, q_2, \ldots\}$ the query workload. The problem is then:

$$\begin{aligned} \text{minimize:} \quad & \sum_{q \in Q} F(B, q) \\ \text{subject to:} \quad & \sum_i b_{ij} |L_i| \leq S \\ & b_{ij} \in \{0, 1\} \end{aligned}$$

Although the problem looks daunting, the cost function F has extra structure which allows us to find a near optimal solution. Consider the benefit of an extra bitmap, b_{ij}, when a previous set B has already been selected. This is exactly $F(B \cup \{b_{ij}\}, q) - F(B, q)$. Contrast this with the extra benefit when a larger set, $\hat{B} \supseteq B$ has already been selected, $F(\hat{B} \cup \{b_{ij}\}, q) - F(\hat{B}, q)$. The cost function defined in Section 3.1 exhibits diminishing returns, that is, the extra benefit of b_{ij} can only decrease as other bitmaps are added. More formally, the function is submodular.

LEMMA 1. *Let F be the cost function as above, and $B \subseteq \hat{B}$ be two sets of bitmaps already selected. Then for any b_{ij}, $F(\hat{B} \cup \{b_{ij}\}, q) - F(\hat{B}, q) \leq F(B \cup \{b_{ij}\}, q) - F(B, q)$.*

Since the sum of submodular functions is itself submodular, we know that the objective in the mathematical problem defined above is also submodular. We now appeal to the theory of submodular function maximization subject to a budget constraint. As [15] shows, the simple greedy algorithm achieves a constant factor approximation to the optimum. At every iteration, for every potential bitmap b_{ij}, the algorithm computes the ratio of the benefit to the increase in index size:

$$\lambda_{ij} = \frac{\sum_{q \in Q} F(B \cup \{b_{ij}\}, q) - F(B, q)}{|L_i|}. \tag{1}$$

Then b_{ij} with the maximum λ_{ij} is selected as the next bitmap to be added, and the benefit ratios λ_{ij} are recomputed.

LEMMA 2 ([15]). *The greedy algorithm finds a set B that forms a $1/2(1 - 1/e) - \epsilon$ approximation to the optimal value of $\sum_{q \in Q} F(B, q)$.*

4.2 Precomputed lists

The problem and the algorithm for selecting precomputed posting lists is similar in spirit to the one for bitmaps. Given a set of precomputed lists $P = \{p\}_{ij}$, where p_{ij} is the indicator variable representing whether the results of query $t_i t_j$ were precomputed, we denote by $F(P, q)$ the cost of evaluating query q given P. We assume q is evaluated using the optimal set of posting lists minimizing the evaluation cost. Adding an extra precomputed list p to P can obviously only reduce F, but at the cost of storing a new list of size $|L_i \cap L_j|$.

Note that although precomputed lists can encode conjunctions of more than two terms, the utility of precomputed lists drops sharply with each new term, as the list requires that *all* of the terms be present in the query. We found that in the AOL query log (described in Section 6) the 100 most frequent three-term conjunctions appear in only one quarter of the 100 most frequent two-term conjunctions. Moreover, since the number of possible multi-term conjunctions grows exponentially with the number of terms in a query, considering them during index construction and during query evaluation becomes computationally expensive. We thus consider only precomputed lists of two terms that capture most of the benefit of such lists.

We can again describe the optimization problem as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{q \in Q} F(P, q) \\ \text{subject to} \quad & \sum_{ij} p_{ij} |L_i \cap L_j| \leq S \\ & p_{ij} \in \{0, 1\} \end{aligned}$$

An argument similar to the one in the last section shows that the objective function remains submodular and thus the same submodular function optimization technique described in [15] can be applied to quickly obtain a constant approximation to the optimum solution. Given a set of already selected precomputed lists P, we sort the potential precomputed lists by:

$$\lambda'_{ij} = \frac{\sum_{q \in Q} F(P \cup \{p_{ij}\}, q) - F(P, q)}{|L_i \cap L_j|}, \tag{2}$$

and select the precomputed list p_{ij} that maximizes λ'_{ij}.

LEMMA 3 ([15]). *The greedy algorithm finds a set P that forms a $1/2(1 - 1/e) - \epsilon$ approximation to the optimal value of $\sum_{q \in Q} F(P, q)$.*

4.3 Hybrid

Unfortunately, we cannot solve the optimization problem for selecting both bitmaps and precomputed list as above. The difficulty arises from the fact that while precomputed lists can carry bitmaps, we can only add bitmaps to the precomputed lists that are already selected. That is, the benefit of adding a bitmap to a precomputed list *increases* (becomes positive) after adding that precomputed list, violating submodularity requirement of the greedy algorithm [15]. Since we are not aware of an alternative optimization technique for achieving constant approximation to the optimum solution for such a problem, we resort to a heuristic approach.

One strategy could be to partition the budget between the two methods, and use each of the above two algorithms to first select precomputed lists and then bitmaps (some of which are added to the precomputed lists). However, deciding upon the budget fraction allocated to precomputed lists and to bitmaps may be hard, as the fraction depends on the distribution of the posting list lengths as well as on the query workload.

We thus use a hybrid algorithm that in each step selects a precomputed list *or* a bitmap that maximizes marginal benefit relative to the precomputed lists and bitmaps selected so far. That is, at each step we select either b_{ij} or p_{ij} that has the maximum marginal benefit given by Equations 1 and 2. To make the marginal benefit functions λ_{ij} and λ'_{ij} directly comparable, they have to be normalized according

on the number of bits used for each bitmap and posting in precomputed lists. Let the number of bits per posting used for a bitmap be β_1 (naturally, $\beta_1 = 1$), and the number of bits per posting in a precomputed list be β_2 (β_2 is the size of the $\langle$*docid, payload* $\rangle$ tuple, in our experiments $\beta_2 = 32$ bits). Then, we divide λ_{ij} by β_1 and λ'_{ij} by β_2.

5. QUERY EVALUATION

Given a query q and its matching posting lists our goal is to select a subset $\mathcal{L}$ of the lists to use for query evaluation. When there are no precomputed lists or bitmaps present, this algorithm has no choice but to use one list per query term. However, when additional information in the form of bitmaps or precomputed lists is available some lists may not be necessary.

5.1 Bitmaps

In the case of bitmaps, we can formally state the problem facing the algorithm as follows. We are given a set of query terms $t_1, t_2, \ldots, t_n$ and a set of postings lists associated with each term $L_1, L_2, \ldots, L_n$. In addition, the algorithm has access to the association matrix B where $b_{ij} = 1$ if there is a bit for term t_j in list L_i's bitmap. Our goal is to find a subset of the lists that minimizes the query cost, yet covers all of the terms. Formally, let $\mathcal{L} \subseteq \{L_1, L_2, \ldots, L_n\}$ be the set of lists that we will be using for query evaluation. $\mathcal{L}$ covers the query q if and only if:

$$\forall t_i \in q, \exists j \text{ such that } L_j \in \mathcal{L} \wedge (b_{ji} = 1 \vee j = i)$$

Then the goal is to find $\mathcal{L}$ that covers q and minimizes $F(B, q)$. Finding the optimum subset of lists is an NP-complete problem, which follows via a simple reduction from the set cover problem. We therefore use a greedy algorithm to select the best subset of the lists (Algorithm 2).

Algorithm 2 Query rewrite algorithm using bitmaps

```
1: Assume that |L_1| ≤ |L_2| ≤ ... ≤ |L_n|
2: Unmark terms t_1, t_2, ..., t_n
3: L ← ∅
4: for i ← 1 to n do
5:    if t_i is unmarked then
6:       L ← L ∪ {L_i}
7:       Mark t_i
8:       for j ← i + 1 to n do
9:          if b_ij = 1 then
10:            Mark t_j
11: return L
```

The algorithm begins with all of the query terms unmarked. It proceeds by examining lists in increasing order of their lengths, since our cost function (Section 3.1) has a monotone dependence on the length of the lists – processing longer lists takes more time. If a given list L is unmarked, it is marked and added to $\mathcal{L}$. The algorithm then checks to see if any of the longer unmarked lists are present in the bitmap of L (and can hence be evaluated using only the information in L). It marks all of the lists that satisfy this condition and continues with the next shortest unmarked list. Although this greedy heuristic carries no formal optimization guarantees, it performed well in practice (see Table 1).

5.2 Precomputed lists

In the case of precomputed lists, the approach is nearly identical to the one described in the previous section. Our goal is to find the set of lists that minimize the cost function and jointly cover all of the query terms. Algorithm 3 begins with all of the query terms unmarked and examines the posting lists in increasing order of their lengths. If a given list for term t_i is unmarked, the algorithm marks it and looks for a precomputed list of term t_i and another unmarked term t_j. If found, the precomputed list L_{ij} is added to $\mathcal{L}$, otherwise L_i is added.

Algorithm 3 Query rewrite algorithm using precomputed lists

```
1: Assume that |L_1| ≤ |L_2| ≤ ... ≤ |L_n|
2: Unmark terms t_1, t_2, ..., t_n
3: L ← ∅
4: for i ← 1 to n do
5:    if t_i is unmarked then
6:       L ← L_i
7:       Mark t_i
8:       for j ← i + 1 to n do
9:          if p_ij = 1 ∧ t_j is unmarked then
10:            L ← L_ij
11:            Mark t_j
12:            break
13:      L ← L ∪ {L}
14: return L
```

5.3 Hybrid

When both bitmaps and precomputed lists are available, we use a hybrid algorithm that first invokes Algorithm 3 to identify precomputed lists and then invokes Algorithm 2 for removing some of these lists that are covered by bitmaps in shorter lists. The rationale for first looking for precomputed lists is the higher potential benefit of identifying a precomputed list that becomes the shortest list for the query, therefore minimizing $|L_1|$ (the first component of our cost function defined in Section 3.1).

6. EXPERIMENTAL RESULTS

We conducted series of experiments to evaluate our algorithms. All experiments were performed on a Linux-based 8-core 1.8GHz server with 16GB memory. Our index construction and retrieval algorithms were implemented as single-threaded Java applications. We report *in memory* list access latencies measured after query rewrite and after preloading all posting lists into memory, averaged over several runs. We focus on the evaluation of in memory indexes since the strict requirements of high-throughput and low-latency, combined with the fact that today's commodity server machines have main memory that can exceed the disk capacities of a decade ago, makes disk-based indexes rarer for the large scale applications such as web search [9] or online advertising platforms that are the focus of our work. Although typical indexes are much larger than the memory capacity of a single machine, the indexes are typically divided into many partitions (shard) so that each partition fits into memory. The intent of our experiments is to highlight the benefits of precomputation in a single index partition.

We indexed the TREC WT10g corpus consisting of 1.68

million web pages. We extracted the textual content of the documents and discarded HTML tags. We built an inverted index where each posting contains a *docid* of four bytes and variable size payload containing bitmaps. Bitmap sizes vary from 0 to 32 bits, rounded up to the next byte boundary. The size for the basic index (with no precomputed lists or bitmaps) was 1.5GB. In our experiments we allowed for precomputation budget of up to three times the index size, which means that our largest index of 6GB (4.5GB occupied by precomputed results) still fits in memory.

The average query latency we measure directly translates to hardware costs of serving a query stream. The average latency of a query over the original index was 2.5 milliseconds. This latency corresponds to 1 on the y-axis in the figures showing query latency.

For query workload, we used the AOL query log [20]. We sorted all queries according to their timestamps and discarded queries containing non-alphanumeric characters, as well as all additional information contained in the log beyond query strings. The resulting 23.6M queries were split into training and testing sets. The training set consists of the first 21M queries from the AOL log, spanning 2.5 months. The testing set is a sample of 50K queries from the remaining 2.6M queries, spanning the following two weeks. The training set was used for creating indexes while the testing set was used for query evaluation.

Using the (properly anonymized) AOL query log has become very common in the research community, despite the controversy surrounding its release and subsequent withdrawal, as it is the largest and the most recent of the publicly available query logs. To make our results reproducible, we have chosen to use the AOL log, and not a proprietary query log.

6.1 Bitmaps and precomputed lists

We first examine the effects of each of the two precomputation techniques alone and then study their combination. We allowed for a memory budget equal to the size of 25% of the original index, for precomputed results. We then evaluated the benefit of allocating this budget for (1) bitmaps only, (2) precomputed lists (the baseline), and (3) both bitmaps and precomputed lists using the hybrid algorithm (Section 4.3). The ratio between the average query latency when using the index with precomputed results and the average latency using the original index is depicted in Figure 3. The figure shows that precomputed lists alone reduce the average query latency by 32% while the bitmaps alone by 41%. A combination of the two techniques achieves an even higher latency reduction of 53%.

We note that the above is not a completely fair comparison, as our algorithm only allows for pairwise intersection lists, while a bitmap may contain information about multiple terms. In order to demonstrate that this is not a limitation, we observe that going from pairwise intersections to triples does *not* reduce the size of the most frequently accessed lists significantly. The total number of postings in the lists for the top 1000 most frequent bigrams is 173.9 million, whereas the total number of postings in the top 1000 trigram lists is 169.9 million, a reduction of only 2.4%. At the same time, the top 1000 trigram lists cover less than one third of the queries covered by the top 1000 bigram lists. Thus, not allowing for larger intersection lists is not a major limitation. Furthermore, this data suggests that the top trigrams have a significant overlap with each other, and, we find this to be the case. Of the top 1000 trigrams 92.5% share a bigram with at least one other trigram in the list. This is exactly the situation where the bitmap approach is superior to simple intersection lists.

We next evaluate two strategies of allocating the shared memory budget for bitmaps and precomputed lists: (1) allocating a fixed fraction of the memory budget for bitmaps and precomputed lists, first selecting precomputed lists and then bitmaps using algorithms described in sections 4.1 and 4.2; and (2) bitmaps and precomputed lists simultaneously using the hybrid algorithm (Section 4.3). The ratio between the average query latency when using the index with precomputed results and the average latency using the original index is depicted in Figure 4. It is evident that the hybrid index construction algorithm successfully finds the optimal allocation without the need to decide on how to partition the memory budget between bitmaps and precomputed lists.

Finally, we use the index constructed using the hybrid algorithm and compare the fraction of queries that benefit from at least one precomputed list or at least one bitmap as a function of query MRIS (defined in Section 3.2). Figure 5 shows that while queries with low MRIS are effectively optimized using precomputed lists, as MRIS increases their effectiveness drops to almost zero. The effectiveness of bitmaps, on the other hand, is independent of the MRIS, thus their marginal benefit (captured by the hybrid algorithm) rises with MRIS.

6.2 Memory budget

We next evaluate the marginal benefit of allocating memory for precomputed results using the hybrid algorithm. Figure 6 shows the average query latency as a function of the precomputation budget, from 0% (the original index without precomputation) to 300% (precomputed results occupy 3/4 of the index). The figure shows that allocating as little as 3% of additional memory for precomputed results decreases average query latency by 25%. As the index grows to twice the original size, the latency decreases by almost 2/3. The benefit of further increasing precomputation budget is relatively low.

Note that although we experimented with encoding term co-occurrences with single bits ($\beta_1 = 1$), the results suggest that using multiple-bit encodings (e.g., to indicate terms proximity or term weights) would yield lower but still favorable index size vs. latency tradeoff. For example, for $\beta_1 = 4$ (i.e., 4 bits instead of 1), the index size cost of achieving 25% latency decrease would grow by the factor of 4, that is, the index would increase by 12% only.

6.3 Coverage of new queries

Many query streams, such as user queries to search engines, follow a "heavy tail" distribution – queries that have never been seen before comprise as much as half of all arriving queries [2]. It is thus important to not overfit the training data. To evaluate the effect of precomputation on long tail queries, we identify all queries in the test set that did not appear in the training set. These queries comprise 46% of our test set. Figure 7 shows the latency of all queries and compares it to that of the long tail queries, with and without precomputation. Note that without precomputation the average latency of long tail queries is lower by 22% than the average latency of all queries, since the posting lists

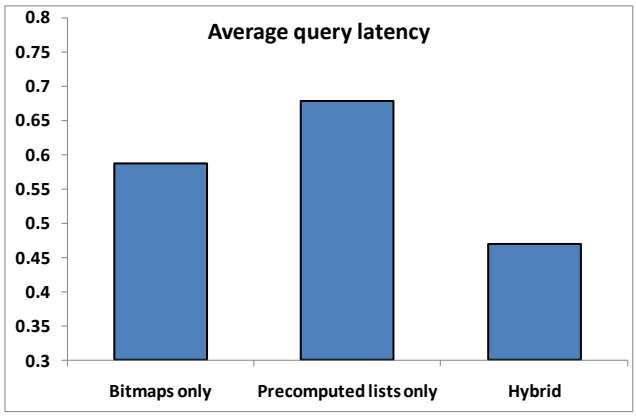

Figure 3: Query latency reduction with different precomputation techniques. Precomputation budget is 25% the size of the original index size.

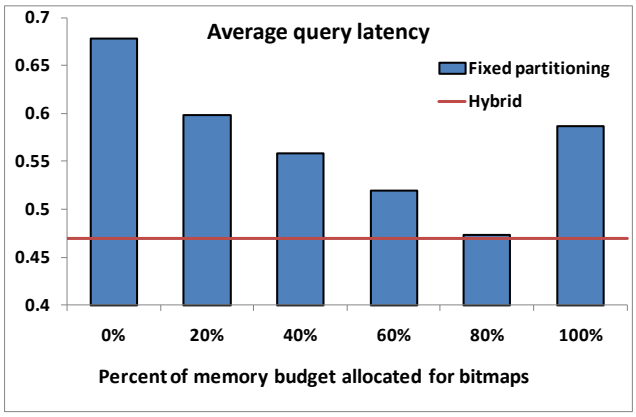

Figure 4: Query latency reduction as a function of memory sharing between bitmaps and precomputed lists. Total precomputation budget is 25% the size of the original index size.

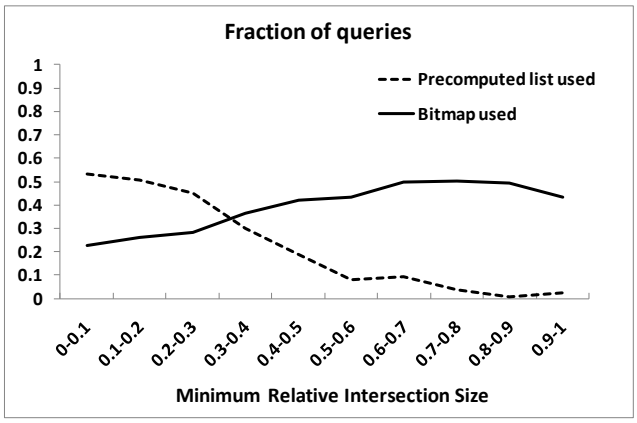

Figure 5: Fraction of queries optimized by at least one precomputed list or one bitmap, by query MRIS.

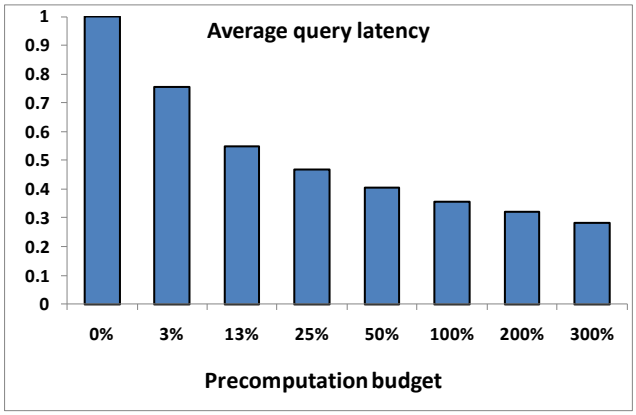

Figure 6: Query latency reduction with increasing precomputation budget (in percents of the original index size).

of terms in long tail queries tend to be shorter than average. This could possibly happen due to higher fraction of rare terms (e.g., names, foreign words, misspellings) in these queries. Yet, although the potential benefit from precomputation is lower for the long tail queries, and despite these exact queries did not appear in the training set, precomputation reduces their average latency by 33%.

To further demonstrate that precomputed results do not overfit the training we fix index and examine the average query latency of AOL queries spanning two weeks. Figure 8 shows that the average latency does not change, indicating that in this timeframe, while the query mix might have changed, the common subqueries remain the same and thus there is no degradation in performance. We conclude that for web search query logs, precomputation successfully captures common subqueries submitted in the future.

6.4 Query rewrite performance

We next evaluate how well the greedy query rewrite algorithm performs compared to the optimal algorithm. We identify the optimal query rewrite by evaluating our cost function on all possible rewrites given the index and selecting the one with the lowest cost. Table 1 summarizes the results for different precomputation budgets ("Avg. lists" stands for the average number of posting lists used to compute query results). Expectedly, as the precomputation budget increases, the quality of our heuristic approximation goes down, albeit only slightly. Even when half of the index contains precomputed results, the evaluation cost due to heuristic approximation is only two percent higher than with the optimal rewrite.

Metric \ Budget	3%	25%	100%
Avg. lists before rewrite	2.83	2.83	2.83
Avg. lists after heuristic rewrite	2.49	2.13	1.92
Avg. lists after optimal rewrite	2.48	2.12	1.90
Fraction of rewritten queries	28%	50%	60%
Heuristic rewrite is optimal	97%	91%	87%
Heuristic cost over optimal	0.2%	0.8%	1.7%

Table 1: Query rewrite performance for precomputation budget of 3%, 25%, and 100% of the original index size.

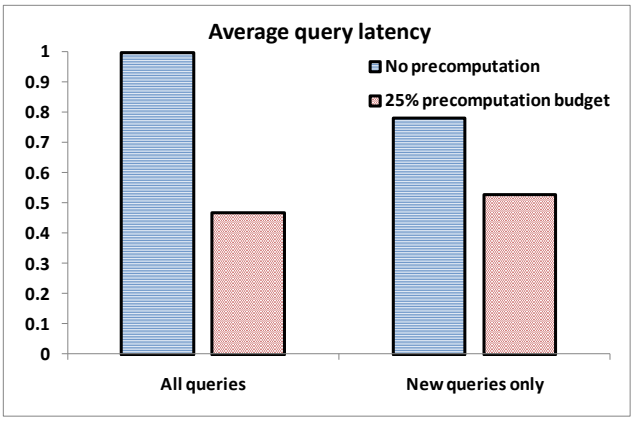

Figure 7: Query latency reduction of long tail queries compared to all queries. Precomputation budget is 25% of the original index size.

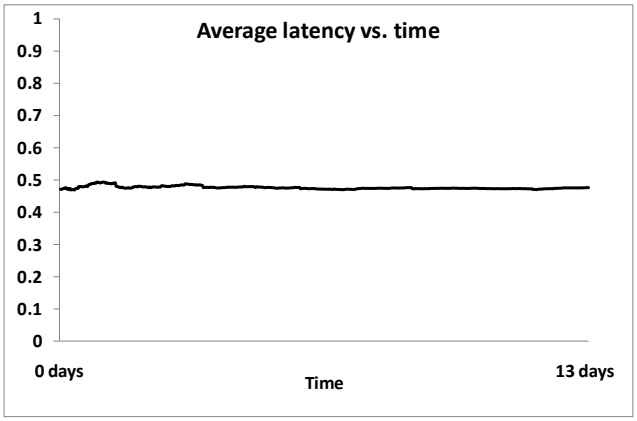

Figure 8: Query latency reduction as a function of time after the last query appearing in the training set. Precomputation budget is 25% of the original index size.

6.5 Index construction overhead

While in this work we focus on query evaluation costs, and not on efficient index construction, we report index construction overhead using straightforward batch implementation of the algorithms we describe. We used a single-threaded code that used up to 16G of memory.

Recall that the TREC WT10g corpus we index consists of 1.68 million documents of average length 217 (after discarding non-textual content), corresponding to 365 million postings. Index inversion takes about 50 minutes. For selecting bitmaps, precomputed lists, and the hybrid approach we used the same code that could be configured to implement either approach. For precomputation budget of 25%, the analysis of the query log (21M training queries) and the greedy submodular optimization take about 160 minutes, while adding the selected precomputation lists and bitmaps to the index takes additional 13 minutes. Thus, the total overhead of adding precomputed results to the index is of the same order as the time it takes to construct the standard inverted index. Obviously, for a partitioned index, index construction can be parallelized across partitions.

7. RELATED WORK

To the best of our knowledge, this is the first work that uses the posting list payloads to encode term co-occurrence. Several other works explored using payloads for different kinds of precomputation. In [25] the authors propose to index phrase queries using a combination of a nextword and a phrase index, achieving significant reduction in the query evaluation latency at the expense of small increase of the index size. In [11] the authors explore the use of the posting payloads to improve processing of phrase queries by using algebraic signatures of the content preceding the current posting. In [27] the authors propose to precompute and store in the index term proximity information, and then use it to speed-up retrieval.

There is an extensive line of works on list intersection techniques, [4, 5, 8, 10] to name a few. These works mainly focus on the worst-case complexity of computing intersections of arbitrary lists. Conversely, in this paper our goal is optimizing query performance for the *given* index and query workload, exploiting the fact that some lists are more likely to be intersected than the others.

One of the first works that explored the use of precomputed posting lists was done by Long and Suel [18]. They proposed a three-level caching strategy, where the second level consisted of precomputed posting lists of frequently occurring pairs of terms. A similar strategy was also proposed in the context of P2P search [22]. In [16], the authors show how to use precomputed posting lists together with early termination in order to improve the query performance of information retrieval systems. The contribution of that work is combining these two techniques using rigorous theoretical analysis. In addition, the authors performed empirical tests on the TREC GOV2 data set and on real web queries showing the performance gains of their early termination method.

Several authors have focused on improving top-k query performance through optimized DAAT and TAAT query evaluation algorithms (e.g. [7, 23]). These algorithms focus on vector space model queries and use upper bounds on term weights to *skip* posting entries that are guaranteed to not influence query results. Turtle and Flood [23] propose both TAAT and DAAT versions of the *max score* algorithm. Broder et. al [7] propose the WAND (Weighted AND) DAAT algorithm, which uses upper bounds to reduce the number of full score computations. In [1] weight quantization is used to reduce the evaluation cost at the expense of slight decrease in accuracy.

Another related line of work is the use of *fancy lists* to improve query performance [17, 21]. Fancy lists are small posting lists that contain the documents with the highest score for each term in the dictionary. In [21], for instance, a new DAAT algorithm that improves upon the DAAT *max score* [23] is proposed. This algorithm uses fancy lists to set a better initial threshold for DAAT *max score*.

8. CONCLUSIONS

In this paper we introduced the concept of bitmaps for optimizing query evaluation over inverted indexes. Bitmaps allow for a flexible way of storing information about term co-occurrences and complement the traditional approach of precomputed lists. We proposed a greedy procedure for the

problem of selecting bitmaps and precomputed lists that is a constant approximation to the optimal algorithm. The analysis of bitmaps and precomputed lists over the TREC WT10g corpus shows that the hybrid approach achieves 25% query performance improvement for 3% growth in index size and 71% for 4-fold index size increase. An interesting future work is exploring the application of bitmaps to non-Boolean top-k retrieval. In this case one needs to store extra information in the bitmaps, such as weights in the case of vector space queries and term positions in the case of phrase queries.

9. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, pages 35–42, 2001.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *Proc. 30th SIGIR*, pages 183–190, 2007.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[4] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *CPM*, pages 400–408, 2004.

[5] J. Barbay, A. L?pez-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Experimental Algorithms*, volume 4007 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin / Heidelberg, 2006.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW*, 1998.

[7] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[8] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *SPIRE*, pages 137–148, 2007.

[9] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, page 1, 2009.

[10] B. Ding and A. C. König. Fast set intersection in memory. *Proc. VLDB Endow.*, 4:255–266, January 2011.

[11] C. du Mouza, W. Litwin, P. Rigaux, and T. Schwarz. As-index: a structure for string search using n-grams and algebraic signatures. In *CIKM 2009*, pages 295–304, New York, NY, USA, 2009. ACM.

[12] M. Fontoura, E. J. Shekita, J. Y. Zien, S. Rajagopalan, and A. Neumann. High performance index build algorithms for intranet search engines. In *VLDB*, 2004.

[13] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[14] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8), 2003.

[15] A. Krause and C. Guestrin. A note on the budgeted maximization of submodular functions. Technical Report CMU-CALD-05-103, Carnegie Mellon University, 2005.

[16] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *WSDM 2009*, pages 222–231, New York, NY, USA, 2009. ACM.

[17] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, 2003.

[18] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW 2005*, pages 257–266, New York, NY, USA, 2005. ACM.

[19] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *WWW*, 2001.

[20] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *Proc. 1st InfoScale*, 2006.

[21] T. Strohman, H. R. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, pages 219–225, 2005.

[22] B. B. Sudarshan, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *In Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems*, 2003.

[23] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6), 1995.

[24] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing boolean expressions. *PVLDB*, 2(1):37–48, 2009.

[25] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[26] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[27] H. Yan, S. Shi, F. Zhang, T. Suel, and J.-R. Wen. Efficient term proximity search with term-pair indexes. In *Proc. 19th CIKM*, pages 1229–1238, 2010.

[28] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.