World Wide Web (2006) 9: 369–395
DOI 10.1007/s11280-006-0221-0

Three-Level Caching for Efficient Query Processing in Large Web Search Engines

Xiaohui Long · Torsten Suel

Received: 7 November 2005 / Revised: 4 April 2006 / Accepted: 13 April 2006
© Springer Science + Business Media, LLC 2006

Abstract Large web search engines have to answer thousands of queries per second with interactive response times. Due to the sizes of the data sets involved, often in the range of multiple terabytes, a single query may require the processing of hundreds of megabytes or more of index data. To keep up with this immense workload, large search engines employ clusters of hundreds or thousands of machines, and a number of techniques such as caching, index compression, and index and query pruning are used to improve scalability. In particular, two-level caching techniques cache results of repeated identical queries at the frontend, while index data for frequently used query terms are cached in each node at a lower level. We propose and evaluate a three-level caching scheme that adds an intermediate level of caching for additional performance gains. This intermediate level attempts to exploit frequently occurring pairs of terms by caching intersections or projections of the corresponding inverted lists. We propose and study several offline and online algorithms for the resulting weighted caching problem, which turns out to be surprisingly rich in structure. Our experimental evaluation based on a large web crawl and real search engine query log shows significant performance gains for the best schemes, both in isolation and in combination with the other caching levels. We also observe that a careful selection of cache admission and eviction policies is crucial for best overall performance.

Keywords web search · search engine architecture · search engine query processing · inverted index · caching

Work supported by NSF CAREER Award CCR-0093400 and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

X. Long · T. Suel (✉)
Department of Computer and Information Science, Polytechnic University,
Brooklyn, NY 11201, USA
e-mail: suel@poly.edu

X. Long
e-mail: xlong@cis.poly.edu

1 Introduction

Due to the rapid growth of the Web from a few thousand pages in 1993 to its current size of many billion pages, users increasingly depend on web search engines for locating relevant information. One of the main challenges for search engines is to provide a good ranking function that can identify the most useful results from among the many relevant pages, and a lot of research has focused on how to improve ranking, e.g., through clever term-based scoring, link analysis, or evaluation of user traces.

Once a good ranking function has been engineered, query throughput often becomes a critical issue. Large search engines need to answer thousands of queries per second on collections of several billion pages. Even with the construction of optimized index structures, each user query requires a significant amount of data processing on average. To deal with this workload, search engines are typically implemented on large clusters of hundreds or thousands of servers, and techniques such as index compression, caching, and result presorting and query pruning are used to increase throughput and decrease overall cost.

To better understand the performance issue, we need to look at the basic structure of current search engines. These engines, like many other information retrieval tools, are based on an *inverted index*, which is an index structure that allows efficient retrieval of documents containing a particular word (or *term*). An inverted index consists of many *inverted lists*, where each inverted list I_w contains the IDs of all documents in the collection that contain a particular word w, sorted by document ID or some other measure, plus additional information such as the number of occurrences in each document, the exact positions of the occurrences, and their context (e.g., in the title, in anchor text).

Given, e.g., a query containing the search terms “apple”, “orange”, and “pear”, a typical search engine returns the 10 or 100 documents that score highest with respect to these terms. To do so, the engine traverses the inverted list of each query term, and uses the information embedded in the inverted lists, about the number of occurrences of the terms in a document, their positions and context, to compute a score for each document containing the search terms. In addition, scores based on link analysis or user feedback are often added into the total score of a document; in most cases this does not affect the overall structure of the computation if these contributions can be precomputed offline (e.g., using Page Rank).

Clearly, each inverted list is much smaller than the overall document collection, and thus scanning the inverted lists for the search terms is much preferable to scanning the entire collection. However, the lengths of the inverted lists grow linearly with the size of the collection, and for terabyte collections with billions of pages, the lists for many commonly used search terms are in the range of tens to hundreds of megabytes or even more. Thus, query evaluation is expensive, and large numbers of machines are needed to support the query loads of hundreds or thousands of queries per second typical of major engines. This motivates the search for new techniques that can increase the number of queries per second that can be sustained on a given set of machines, and in addition to index compression and query pruning, caching techniques have been widely studied and deployed.

Caching in search engines has been studied on two levels [33]. The first level of caching, *result caching*, takes place at the frontend, and deals with the case where identical queries are issued repeatedly by the same or different users. Thus, by keeping a cache of a few ten thousand to a few million results that have recently been returned by the engine, we can filter repeated queries from the workload and increase overall throughput. Result caching has been studied in [22, 23, 26, 33, 39]. It gives a measurable benefit at a low cost (each result could simply be stored as a complete HTML page in a few KB), though the benefit is limited by the degree of repetition in the input stream. At a lower level, *list caching* is used to keep inverted lists corresponding to frequently used search terms in main memory, resulting in additional benefits for engines with disk-based index structures. The benefits of a two-level caching approach in an actual search engine were studied in [33].

In this paper, we propose and evaluate a three-level caching architecture with an additional intermediate level of caching. This level, called *intersection caching* or *projection caching* (depending on the implementation), caches inverted list data for pairs of terms that commonly occur together in queries with more than two search terms. The basic idea is very simple and relies on the fact that all of the major search engines by default only return documents that contain *all* of the search terms. This is in contrast to a lot of work in the IR community where every document containing at least one of the terms participates in the ranking; we will discuss this issue again later. Thus, search engines need to score only those documents that occur in the intersection of the inverted lists. Unfortunately, in most cases the most efficient way to find the intersection still involves a complete scan over the lists, and this dominates the cost of query processing. By caching pairwise intersections between lists, which are typically much smaller than each of the two lists, we hope to significantly reduce this cost in subsequent queries. We note that the basic idea of caching intersections was also recently proposed in the context of P2P-based search in [7], but the scenario and objectives are rather different as discussed later.

While the idea of caching intersections is very simple, the resulting *weighted caching* problem turns out to be quite challenging. In the main technical part of the paper, we discuss and evaluate several online and offline caching algorithms. Even very restricted classes of the problem are NP-Complete, but we show that there are practical approaches that perform much better than the basic *Landlord* algorithm [11, 40] for weighted caching on typical query traces. We also perform an evaluation of the performance of all three caching levels together. The conclusion is that caching gives a significant overall boost in query throughput, and that each level contributes measurably.

The next section gives some technical background, and Section 3 discusses related work. The three-level caching approach is described and discussed in detail in Section 4. Section 5 studies the resulting intersection caching problem and presents two basic approaches. Section 6 refines these approaches and performs a detailed experimental evaluation across all three caching levels. Finally, Section 7 provides some concluding remarks.

2 Search engine query processing

In this section, we provide some background on query execution in search engines. We assume that we have a document collection $D = \{d_0, d_1, \ldots d_{n-1}\}$ of n web pages that have already been crawled and are available on disk. Let $W = \{w_0, w_1, \ldots, w_{m-1}\}$ be all the different words that occur anywhere in the collection. Typically, almost any text string that appears between separating symbols such as spaces, commas, etc., is treated as a valid word (or *term*) for indexing purposes in current engines.

Indexes An *inverted index I* for the collection consists of a set of inverted lists $I_{w_0}, I_{w_1}, \ldots, I_{w_{m-1}}$ where list I_w contains a *posting* for each occurrence of word w. Each posting contains the ID of the document where the word occurs, the (byte or approximate) position within the document, and possibly information about the context (in a title, in large or bold font, in an anchor text) in which the word occurs. The postings in each inverted list are often sorted by document IDs, which enables compression of the list. Thus, Boolean queries can be implemented as unions and intersections of these lists, while phrase searches (e.g., "New York") can be answered by looking at the positions of the two words. We refer to [38] for more details.

Queries A query $q = \{t_0, t_1, \ldots, t_{d-1}\}$ is a set of terms (words). For simplicity, we ignore search options such as phrase searches or queries restricted to certain domains at this point. In our caching problems we are presented with a long sequence of queries $Q = q_0, q_1, \ldots, q_{l-1}$, where $q_i = \{t_0^i, t_1^i, \ldots, t_{d_i-1}^i\}$.

Term-based ranking The most common way to perform ranking in IR systems is based on comparing the words (terms) contained in the document and in the query. More precisely, documents are modeled as unordered bags of words, and a ranking function assigns a score to each document with respect to the current query, based on the frequency of each query word in the page and in the overall collection, the length of the document, and maybe the context of the occurrence (e.g., higher score if term in title or bold face). Formally, a *ranking function* is a function F that, given a query $q = \{t_0, t_1, \ldots t_{d-1}\}$, assigns to each document D a score $F(D, q)$. The system then returns the k documents with the highest score. One popular class of ranking functions is the *cosine measure* [38], for example

$$F(D, q) = \sum_{i=0}^{d-1} \frac{w(q, t_i) \cdot w(D, t_i)}{\sqrt{|D|}},$$

where $w(q, t) = \ln(1 + n/f_t)$, $w(D, t) = 1 + \ln f_{D,t}$, and $f_{D,t}$ and f_t are the frequency of term t in document D and in the entire collection, respectively. Many other ranking functions have been proposed, and the techniques in this paper are not limited to any particular class.

AND vs. OR Many ranking function studied in the IR community, including the above cosine measure, do not require a document to contain all query terms in order to be returned in the results. (E.g., a document containing two out of three query terms multiple times or in the title may score higher than a document containing all three terms.) However, most search engines enforce `AND` semantics for queries and only consider documents containing all query terms. This is done for various reasons involving user expectations, collection size, and the preponderance of short queries (thus, for most queries, there will be many documents containing all query terms). Our approach fundamentally depends on `AND` semantics, which are the default in essentially all major engines (e.g., Google, Yahoo, MSN).

Query execution Given an inverted index, a query is executed by computing the scores of all documents in the intersection of the inverted lists for the query terms. This is most efficiently done in a *document-at-a-time* approach where we simultaneously scan the inverted lists, which are usually sorted by document ID, and compute the scores of any document that is encountered in all lists. (It is shown in [21] that this approach is more efficient than the *term-at-a-time* approach where we process the inverted lists one after the other.) Thus, scores are computed *en passant* while materializing the intersection of the lists, and top-k scores are maintained in a heap structure. In the case of AND semantics, the cost of performing the arithmetic operations for computing scores is dominated by the cost of traversing the lists to find the documents in the intersection, since this intersection is usually much smaller than the complete lists.

Search engines use a number of additional factors not present in standard cosine-type ranking functions, such as context (e.g., term occurs in title, URL, or bold face), term distance within documents (whether two terms occur close to each other or far apart in the text), and link analysis and user feedback. The first two factors can be easily included while computing scores as outlined above. The most commonly used way to integrate the other factors is to precompute a global importance score for each document, as done in PageRank [9], or a few importance scores for different topic groups [19], and to simply add these scores to the term-based scores during query execution [25, 30, 31]. Our approach does not depend on the ranking function as long as the total cost is dominated by the inverted list traversal.

Search engine architecture Major search engines are based on large clusters of servers connected by high-speed LANs, and each query is typically executed in parallel on a number of machines. In particular, current engines usually employ a *local index organization* where each machine is assigned a subset of the documents and builds its own inverted index on its subset. User queries are received at a frontend machine called *query integrator*, which broadcasts the query to all participating machines. Each machine then returns its local top-10 results to the query integrator to determine the overall top-10 documents [22].

Each subset of the collection is also replicated and indexed on several nodes, and multiple independent query integrators can be used. We note that there are alternative partitioning approaches such as the *global index organization* and various hybrids that are not commonly used in large engines though they may have advantages in certain scenarios; see [4, 28, 36, 37] for discussion.

Query processing optimizations Given simple mechanisms for load balancing and enough concurrency on each machine, the local index organization results in highly efficient parallel processing. Thus, the problem of optimizing overall throughput reduces again to the single-node case, i.e., how to maximize the number of queries per second that can be processed locally on each machine with a reasonable response time. One commonly used technique is to compress each inverted list using various coding techniques [38], thus reducing overall I/O for disk-based index structures but increasing CPU work. Because of this tradeoff, fairly simple and fast techniques tend to outperform schemes geared towards optimal compression [34]. Our experiments use compression but do not depend on it.

A second optimization attempts to determine the top-k results without a complete scan of the intersection or union of the inverted lists, by presorting the lists according to their contributions to the score and terminating the traversal early (or by removing low-scoring postings from the index altogether [15]). There has been a significant amount of work in the IR and database communities on this issue under various scenarios; see [1, 2, 12, 14, 16, 25, 29, 36] for recent work. Various schemes are apparently in use in current engines but details are closely guarded. Note that these techniques are designed for certain types of ranking functions and, e.g., do not easily support use of term distance within documents. Our experiments use a full traversal of the list intersections, but our approach could be adapted to pruned schemes as well though this is beyond the scope of this paper. A third common optimization is caching schemes, discussed in detail in the next section.

3 Discussion of related work

For more background on indexing and query execution in IR and search engines, see [3, 5, 38]. For basics of search engine architecture we refer to [8, 9, 22, 32]. In the following, we focus on previous work on caching and on other issues directly relevant to our work.

Result caching As indicated, result caching filters out repetitions in the query stream by caching the complete results of previous queries for a limited amount of time. It was studied in [22, 23, 26, 33, 39] and is probably in use in most major engines. Result caching can be easily implemented at the query integrator, and [39] also proposes caching results in the internet closer to the user. Work in [22, 23] also connects result caching to the problem of efficiently returning additional result pages for a query, which is most efficiently done by computing and storing more than just 10 results for each query. Result caching only works on completely identical queries and is thus limited in its benefits by the query stream. Published numbers on the percentage of queries that can be answered from the cache range from 30 [26] to 80% [32]. Result caching is easy to implement and does give significant benefits even under very simple caching policies. A side effect of result caching is that the average number of search terms increases for those queries that are actually executed, since single-term and two-term queries are more likely to be already cached.

List caching At the lower level inside each machine, frequently accessed inverted lists are cached in main memory to save on I/O. This is sometimes done transparently by the file system or when using a database system such as Berkeley DB to store the index [28], though for typical IR and web search workloads better results may be achievable with specialized caching policies [20]. Of course, list caching only applies to disk-resident index structures, and some engines attempt to keep all or most of the index in main memory for optimum performance.

Two-level caching In [33], Saraiva et al. evaluate a two-level caching architecture using result and list caching on the search engine *TodoBR*, and show that each level contributes significantly to the overall benefit. For list caching, a simple LRU approach is used. We note that it is possible that techniques similar to ours are already in use in one of the major engines, but this type of information is usually kept highly confidential and we are not aware of it.

Caching in P2P search The basic idea of caching results of intersections that we use in our three-level caching approach was recently also proposed in the context of peer-to-peer search in [7]. We note that the approach in [7] is quite different from ours. Their main goal is to avoid repeated transmissions of inverted list data in a peer-to-peer system with global index organization, while we are interested in improving query throughput in each node by decreasing disk traffic and CPU load. The main emphasis in [7] is on distributed data structures for keeping track of intersections that are cached somewhere in the system, while in our case this problem is easily solved by a standard local data structure. Our emphasis is on the use of intersection caching in a three-level cluster-based architecture, with different algorithms and cost trade-offs than in a peer-to-peer environment, and its performance on a large query load from a real engine. As pointed out in [7], there is also some similarity to views and join indexes in database systems.

Set intersections We note that in some scenarios, there are more efficient ways to intersect two lists than a scan. In particular, when lists are of very different lengths or the docIDs in the lists are clustered, approaches that use forward seeks to skip parts of the lists are often preferable. However, for disk-resident inverted indexes, we only save on disk access times if a forward seek can skip a fairly significant amount of data. Examples of such approaches are the *zig-zag joins* in databases [17] and the recent *adaptive set intersection* techniques in [13]. In addition to savings in disk time, such techniques can also result in decreased CPU costs, and in bandwidth savings in peer-to-peer environments [24].

Optimizations for phrases Caching of intersections is related to the problem of building optimized index structures for phrase queries [6] (i.e., “New York”). In particular, intersections can be used to evaluate phrase queries, while on the other hand some of the most profitable pairs of lists in intersection caching turn out to be common phrases. Note that exhaustive index structures for two-word phrases have only a small constant factor overhead over a standard index, since each occurrence of a word is directly followed by only one other word. Caching all intersections between terms, on the other hand, is impossible and thus appropriate caching policies are needed.

Weighted caching In many caching problems, the benefit of caching an object is proportional to its size (e.g., when caching to avoid disk or network traffic). Weighted caching problems deal with the case where each object has a size and a benefit that may be completely independent of its size. Weighted caching problems are, e.g., studied in [11, 40], which propose and analyze a simple algorithm called *Landlord* that basically assigns leases to objects based on their size and benefit and evicts the object with the earliest expiring lease. Both [11, 40] perform a competitive analysis of the *Landlord* algorithm, and some experimental results for a web caching scenario unrelated to search are given in [11]. In our case, we are dealing with a weighted caching problem where the size of the cached object is the size of an intersection or projection, and the benefit is related to the difference between this size and the sizes of the complete lists. Moreover, there is a cost in inserting an object into the cache, which requires us to employ appropriate cache admission policies. Additional complications arise in our scenario because costs and benefits can be with respect to both disk accesses and CPU costs, and in fact there may be trade-offs between the two as we will see.

4 A three-level caching approach

We now describe and discuss the proposed three-level caching architecture in detail. The architecture is motivated by a few simple observations on available search engine logs. In particular, result caching works very well on single-term and two-term queries but does not perform as well on queries with more terms, which are less likely to be exactly repeated. However, an analysis of large query logs indicates that queries with three or more terms are likely to contain at least one pair of terms that has previously appeared together. Thus, a three-term query $\{a, b, c\}$ could be processed by accessing the inverted list I_a for term a and a cached list for the intersection of I_b and I_c. If the three lists I_a, I_b, I_c are of approximately the same length, and the intersection of I_b and I_c is much smaller than either of the two lists, then we would save almost a factor of 3 even with only one pair having occurred previously. If two pairs have previously occurred, then by scanning the two intersections we could save most of the cost of the query.

We will discuss the exact format and treatment of the cached intersections later. By combining result, intersection, and list caching, we get a three-level caching architecture shown in Figure 1 and summarized as follows:

- **Result caching:** The query integrator maintains a cache of the results of recent queries, either in memory or on disk. Cache size and eviction policy are typically not critical as large numbers of results can be cached cheaply. For our query log of about a million queries, results can be cached essentially over the entire log. Queries not covered by result caching are broadcast to query processing nodes.
- **Intersection caching:** At each node, a certain amount of extra space, say 20 or 40% of the disk space used by the index, is reserved for caching of intersections. These intersections reside on disk and are basically treated as part of the inverted index or as a separate inverted index. For each multi-term query, we check if any pairwise intersections are already cached, and use these to process the query. In addition, during processing we create *en passant* additional

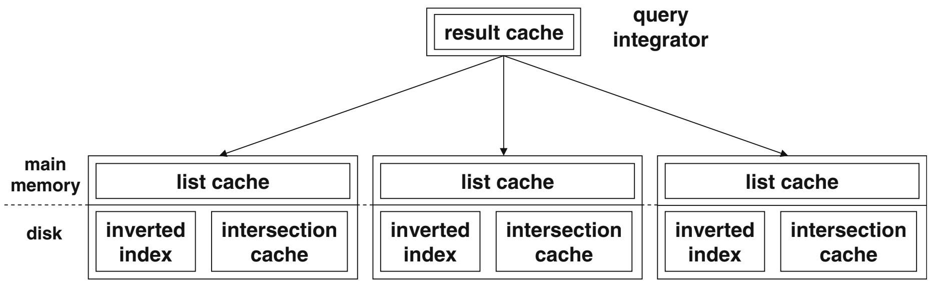


Figure 1 Three-level caching architecture with result caching at the query integrator, list caching in the main memory of each node, and intersection caching on disk.

intersections for some or all of the pairs of terms in the query and add them to the cache. We will show that this can be done very efficiently if we are careful about which intersections are created and added to the cache.

– **List caching:** At the lowest level, a limited amount of main memory (typically at least several hundred MB in nodes with at least one GB of memory) is used to cache frequently accessed inverted lists as well as intersections.

Thus, intersection caching complements result caching as it focuses on queries with three and more terms, and is orthogonal to list caching. Intersection caching is relevant to both disk-based and memory-based index structures, though the performance ramifications are somewhat different as we will see.

4.1 Intersection vs. projection caching

We now discuss the precise format of the cached intersections. Recall that an inverted list is a sequence of postings sorted by document ID, with each posting containing the document ID and additional information about each occurrence of the term in the document. In order to use an intersection instead of the original list during query execution, this data has to be preserved for postings whose document IDs appear in both lists. Thus, a posting in the intersection list would consist of a document ID and information about all occurrences of both words in the document.

However, in our implementation we decided to follow a slightly different approach which we call *projection caching*. As shown in Figure 2, instead of creating the intersection of lists I_a and I_b, we create two projections $I_{a \to b}$ and $I_{b \to a}$, where $I_{a \to b}$ contains all postings in I_a whose document ID also appears in I_b, and $I_{b \to a}$ vice versa. There are several advantages of this approach: (1) Projected inverted lists have exactly the same format as other inverted lists and thus no changes in the query processor are required. Also, creation of projections from complete lists is very simple. (2) $I_{a \to b}$ and $I_{b \to a}$ are treated independently by the list and intersection caching mechanisms and can be evicted separately, which is desirable in some cases. (3) Some additional minor optimizations are possible during query execution; e.g., a query $\{a, b, c\}$ could be executed by using $I_{a \to b}$, $I_{b \to c}$, and $I_{c \to a}$ instead of using pairs. A disadvantage of using projections is that the two projections are larger than a single intersection as the document ID is stored twice. In the case of an inverted index storing only docIDs of postings, this is a significant overhead, but in our case we store docIDs as well as position and context information for each term occurrence, resulting in a smaller relative overhead. We decided to use projections

hello: {<2, 2, 4, 6>, <3, 1, 7>, <7, 2, 1, 6>}

world: {<2, 1, 7>, <5, 2, 1, 4>, <7, 1, 11>}

intersection of “hello” and “world”: {<2; 2, 4, 6; 1, 7>, <7; 2, 1, 6; 1, 11>}

projection$_{\text{"hello"->"world"}}$: {<2, 2, 4, 6>, <7, 2, 1, 6>}

projection$_{\text{"world"->"hello"}}$: {<2, 1, 7>, <7, 1, 11>}

Figure 2 Intersections vs. projections. Here, postings in inverted lists are in the format $\langle docID, freq, pos_0, ..., pos_{freq-1} \rangle$.

in our query processor as the advantages outweigh the slight space penalty. We note that our results can be stated either in terms of intersection or projection caching, and the performance of both schemes is comparable.

4.2 Caching overheads

One common assumption is that caching an object does not result in any cost apart from some limited administrative overhead (data structures) and the space used for caching. In our context, creation of projections for caching is piggybacked onto query execution, and thus (at least at first glance) only involves inverted lists that are being retrieved from disk by the query processor anyway. However, in reality there are some costs associated with creating projections and inserting them into the cache that need to be minimized in order to get good performance. In addition to a very small overhead in making caching and query execution decisions, we have the following more significant costs:

(1) Write cost: Since our projection cache is disk-based, a newly inserted projection has to be written out to disk.
(2) Encoding cost: Before writing out the projection, it is encoded using the same index compression scheme that is used in the inverted index.
(3) Projection creation: Even though projections are created *en passant* during execution of a query, there are certain overheads due to necessary changes in query processing during projection creation.

We report the first cost in our experiments in terms of the number of blocks written out, and show that it can be kept at a fairly low level compared to the savings in read costs. The second cost is typically fairly small, provided that a fast compression scheme is used for the index. In our case, we use a variable-byte compression scheme evaluated in [34], which achieves good compression at a low cost. A more subtle issue is the overhead in creating projections. In our query processor, we use optimized schemes for list intersection, similar to the zig-zag joins in [17], which skip parts of the inverted lists by doing forward seeks. To create a projection during query processing, some of these optimizations have to be switched off, resulting in extra costs, primarily in terms of CPU work. (We observed only small savings in disk access times due to forward seeks, compared to scanning the complete lists, as few of the skips were large enough to justify doing a forward seek on current hard disks.) As we show later, all of these costs can be kept at a very low level by adopting a suitable cache admission policy that prevents the creation of too many projections that are likely to be evicted from cache before being used.

In the first part of our experimental evaluation, we report results in terms of “logical” disk block accesses, including disk reads in query processing and disk writes for adding projections to the cache, but ignoring the caching of lists in main memory. This gives us a rough view of the relative performance of various schemes. In Section 6.2, we then study in detail the CPU savings and overheads due to projection creation, while Section 6.4 evaluates the effect of adding list caching. We note here that the optimal choice of caching policies depends on the relative speeds of disk and CPU, the choice of compression scheme, and whether the index is primarily disk-based or memory-based, and we are unable to evaluate all cases in the limited space. However, we will show that significant performance gains are possible both for primarily disk-based and memory-based index structures, and that the overhead of our schemes is very low.

5 Basic policies for intersection caching

In this section, we study cache maintenance policies for intersection caching. We first define the problem and discuss complexity issues, then present a greedy algorithm for the offline version of the problem, and then describe the *Landlord* algorithm [11, 40] for weighted caching.

5.1 Problem definition and complexity

Recall that we are given a sequence of queries $Q = q_0, q_1, \ldots, q_{l-1}$, where $q_i = \{t^i_0, t^i_1, \ldots, t^i_{d_i-1}\}$. For any query $q = \{t_0, t_1, \ldots, t_{d-1}\}$ that is executed, we can generate and cache any projections $I_{t \to t'}$ with $t, t' \in q$, subject to the maximum cache size C. The size of $I_{t \to t'}$ is $|I_{t \to t'}|$. Query q can be executed by scanning, for each $t \in q$, either I_t or any $I_{t \to t'}$ with $t' \in q$ that is currently in the cache. The cost of executing the query is equal to the sum of the lengths of the lists that are scanned, and our goal is to minimize total query execution cost.

We note that the results in this section can be stated either in terms of intersections or projections. In the offline version of the problem, we assume that the sequence of queries is known ahead of time and that the set of projections in the cache is selected and created before the start of execution. In the online version, queries are presented one at a time and projections can be created and cached during execution of queries as described above and evicted at any point in time. For simplicity, we do not charge for the cost of creating the projections in the above definition, though our later experiments will also consider this issue.

For the offline version, it is not difficult to see that the problem is NP-Complete through a reduction from Subset Sum [18], as are many other caching problems that allow arbitrary object sizes. However, this observation does not really seem to capture the full complexity of our problem. We can strengthen the result as follows.

Theorem 1 *The offline problem is NP-Complete even in the case where all projections are of the same size and queries are limited to at most* 3 *terms.*

Proof sketch By reduction from Vertex Cover [18]. Given a graph $G = (V, E)$ and an integer k, we construct an instance of our caching problem as follows. We have one term t_u for each node in $u \in V$, and in addition we have a special term t'. For each edge $(u, v) \in E$, we create a query $\{t_u, t_v, t'\}$. We assume that all projections between two terms are of the same size (this can be achieved by making all lists disjoint except for a small set of document IDs that appear in all lists), and select a cache size that fits exactly k projections. We also assume that list $I_{t'}$ is significantly larger than all of the I_{t_u}, say $|I_{t'}| > 3 \cdot |E| \cdot |I_{t_u}|$. Then there exists a vertex cover of G of size k iff there exists a selection of cached projections that allows the query trace to be executed with total cost less than $|I_{t'}|$. ■

We note that if all projections are of the same size and queries are limited to 2 terms, then the problem can be solved in polynomial time. On the other hand, the offline problem with 3 terms remains NP-Complete if we allow creation and eviction of projections during query execution, and if we charge a cost for the creation of projections. We discuss the online problem further below.

5.2 A simple greedy algorithm

While in reality we do not have prior knowledge of the query sequence, the offline problem is nonetheless of practical interest since it can be used to make cache space assignments for the future based on analysis of recently issued queries. For this reason, we now describe a simple greedy algorithm for the offline problem, which in each step adds the projection to the cache that maximizes the ratio of additional savings in query processing and projection size. It can be implemented as follows:

(1) For each query q_i in the sequence Q and each projection $I_{t\to t'}$ with $t, t' \in q_i$, create an entry $(t, t', i, |I_{t\to t'}|, |I_t| - |I_{t\to t'}|)$. Note that the last two fields are the size of the projection and the benefit when using it instead of the full list.
(2) Combine all entries with identical t, t' into a single one, but with an additional field at the end, called the total benefit, that contains the sum of the benefits in the combined entries, and with the sequence S of all query numbers i in the combined entries attached to the new entry.
(3) Load the entries into a heap that allows extraction of the element with maximum ratio of total benefit to projection size.
(4) Repeatedly extract an element and add it to the cache. If it does not fit, discard the element and choose another one until the heap is empty. After each extraction of an entry $(t, t', S', *, *, *)$, decrease the total benefit of all projections $(t, t'', S'', *, *, *)$ that $S' \cap S'' \neq \emptyset$.

The size, and thus benefit, of a projection can be efficiently estimated using simple sampling techniques [10] and hence a scan of the inverted lists is not really required. Ignoring these estimation costs, the above algorithm runs in time $O(\kappa \lg(\kappa))$ in the worst case, where $\kappa = \sum_{i=0}^{l-1} |q_i|^2$, i.e., the sum of the squares of the query sizes. In practice, this is a moderate constant times the number of queries since most queries are short.

5.3 The Landlord algorithm

We now consider the online problem. The projection caching problem is an instance of a *weighted caching* problem, where objects have arbitrary sizes and caching of an object results in savings that are independent of (or at least not linear in) their sizes. Such problems have been studied, e.g., in [11, 40], where a class of algorithms called *Landlord* is proposed and analyzed using competitive analysis. We note, however, that our problem comes with an additional twist reminiscent of view selection problems in databases, in that we could use either $I_{a\to b}$ or $I_{a\to c}$ in executing a query $\{a, b, c\}$ but there is little benefit in using both. Thus, a projection for a frequently occurring pair may actually not have much benefit since there may be even better projections available in the cache for most queries where it is applicable. *Landlord* works as follows:

(1) Whenever an object is inserted into the cache, it is assigned a deadline given by the ratio between its benefit and its size.

(2) If another object needs to be evicted to make room for a new one, we evict the element with smallest deadline $d_{\min}$, and deduct $d_{\min}$ from the deadlines of all elements currently in the cache.
(3) Whenever an element in the cache is used, its deadline is reset to some appropriate value discussed later.

A note about Step 2: Instead of deducting $d_{\min}$ from all entries, we can implement this step much more efficiently by summing up all values of $d_{\min}$ that should have been deducted thus far, and taking this sum properly into account during the other steps. This makes the algorithm highly efficient. Also, observe that if in Step 3 deadlines are reset to their original value (ratio between benefit and size), then the algorithm can be seen as a generalization of LRU for weighted caching problems. In [11, 40], the algorithm is shown to be competitive with an optimum solution, but the analysis does not carry over to our problem due to the above "twist". In the following, we will also experiment with several variations of the *Landlord* approach that perform much better on our workload.

6 Experimental evaluation

We now present our experimental setup and give some baseline results for the basic versions of the greedy and *Landlord* algorithms. In Section 6.1 we present and evaluate modified policies with improved performance, and Section 6.2 discusses the CPU overhead of projection creation. Section 6.3 provides additional statistics on the generated projections. Finally, Section 6.4 presents an evaluation over all three levels of caching.

Data sets and experimental setup For our experiments we used a subset of 7.5 million pages selected at random from a crawl of about 120 million web pages crawled by the PolyBot web crawler [35] in October of 2002. This subset size corresponds to a scenario where the pages are evenly distributed over a 16-node search engine, which is a typical setup in our lab. In this case, since projection caching occurs at each individual node, only one machine in the cluster was used. The uncompressed size of the pages was over 100 GB, and after duplicate elimination and indexing we obtained an inverted index structure of size about 10.8 GB. While current commercial engines index several billion pages, these are partitioned and replicated over hundreds or thousands of machines. Moreover, the effects that we measure are all expected to be linear in the size of the document collection, i.e., we would observed the same relative behavior if we scale up both collection size and cache size by the same factor. Thus, we believe that our setup of 7.5 million pages per node provides a realistic evaluation setup.

Queries were taken from a large log of queries issued to the Excite search engine from 9:00 to 16:59 PST on December 20, 1999. For the experiments, we removed $82,506$ queries with words that do not appear in our inverted index of 7.5 million pages. We also removed stopwords from queries. The number of remaining queries is $1,836,248$ with a total of $207,788$ different words, and the average number of words per query is 2.88. We assume a result cache with infinite size on this query log; in general, we expect that result cache size and eviction policy are unlikely to be critical in terms of overall system design and resources.

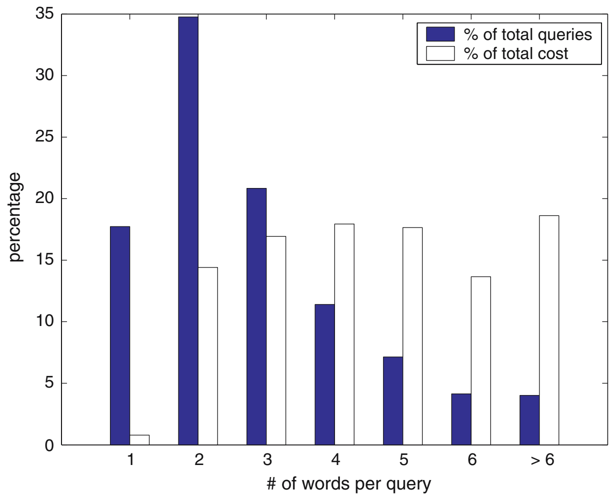
We created three different experimental setups to evaluate query processing costs. Using one setup, we measured the disk access costs of the various policies in terms of the total number of 4 KB disk block accesses. We have found that this provides a reasonable estimate of the cost of our actual query processor [25] when using one or two disks and a fast CPU. We used another setup where we preloaded inverted lists into main memory to measure the CPU costs of the methods, which is important on systems with faster I/O or more main memory for caching. Finally, for the best policies we also integrated projection caching into a real query processor developed in our lab, to measure total system performance including CPU and disk. All experiments were run on a Dell Optiplex 240GX machine with a 1.6 GHz P4, 1 GB of memory, and two 80 GB disks running Linux.

Query characteristics We first look at the distribution of the ratios and total costs for queries with various numbers of terms, by issuing these queries to our query processor with caching completely turned off. In Figure 3, we see that even without result caching, nearly half of the total block access cost was spent on queries with five or more terms, although these queries represent only about 15% of all queries.

Next, we look at how this distribution changes as we filter out repeated queries using result caching. Figure 4 shows the number of queries before and after result caching for different numbers of terms in the query. We see that the number of queries with up to three terms is reduced significantly, while fewer queries with four or more terms are filtered out. Thus, as shown in Figure 5, after result caching an even higher percentage of the total block accesses is spent on queries with four or more terms. In fact, the average cost per remaining query increases from about $2,000$ to almost $2,700$ blocks due to this effect.

To estimate the maximum potential benefit from adding projection caching, we first measured the performance of an ideal projection caching scheme where for every query, all possible projections between terms are created at no cost and inserted into a cache of infinite size. Some results are shown in Figure 6. For example, for queries with 4 words, more than 90% had at least two projections that could be used instead of the complete lists, and more than 35% could be completely

Figure 3 Distribution of frequencies and total costs for queries with different numbers of terms.



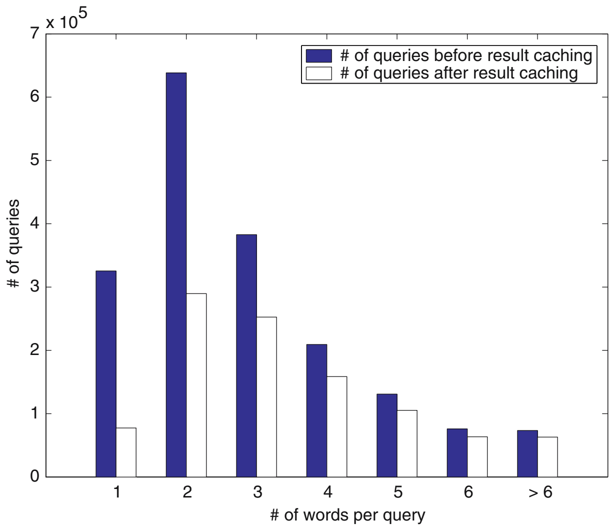


Figure 4 Absolute numbers of k-term queries before and after result caching (in hundreds of thousands).

covered by projections (i.e., for each term there was a projection that could be used instead of the complete list). This effect becomes more pronounced for longer queries, and thus we would expect projection caching to work particularly well in combination with result caching. With such an ideal infinite projection cache, the average cost per query was decreased from 2, 700 blocks to about 950 blocks. The question to be examined next is of course how close we can come to these numbers with realistic projection caching policies that take the cache size and the cost of projection creation into account.

We also considered the potential benefits of using projections with more than two words, i.e., of the type $I_{b \to a \to c}$. Using an ideal infinite projection cache, we observed that the benefit of allowing projections with more than two words appears to be very small. The reason is that while there are many queries for which a three-word

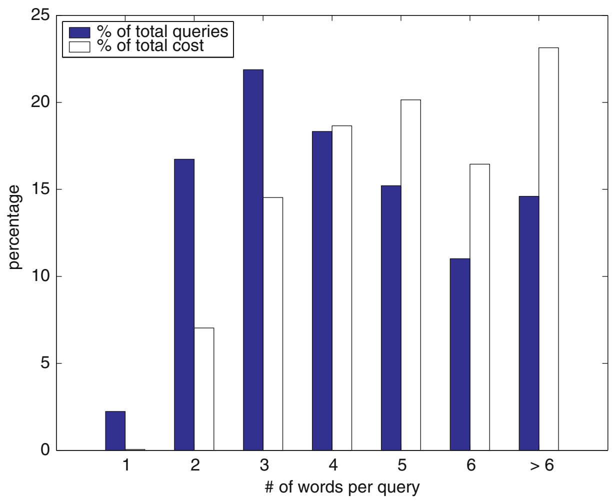


Figure 5 Costs of k-term queries before and after result caching (as percentage of total cost).

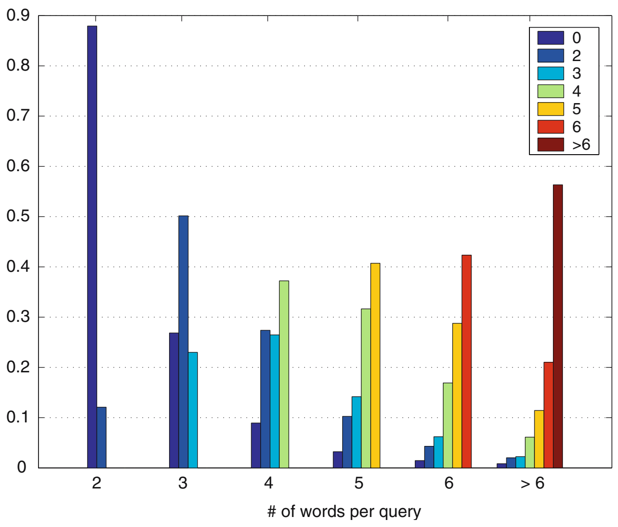


Figure 6 Potential benefits with an ideal infinite projection cache. For queries with k words, we plot the percentage of such queries for which $0, 2, \ldots, k$ projections could be used instead of the complete lists. (Note that for such an ideal cache, if $I_{a \to b}$ is in the cache, then so is $I_{b \to a}$; this means that if we can use at least one projection, than we can in fact use at least two. Also, for each word in a query, only one projection will be used. Therefore, the maximum number of projections used in a k-word query is k.)

projection is potentially applicable (i.e., the words have already occurred together in another query), in these cases there are also several two-word projections that apply. Since two-word projections are already much shorter than the original lists, there is not much of an improvement in absolute terms to be had by using a three-word projection instead. We expect that under a realistic caching policy with finite cache size, the benefit would be even less.

Results for the greedy algorithm We now present results for the basic versions of the greedy and *Landlord* algorithms. In all our experiments, we make sure to "warm up" all levels of caches by running over most of the query log and only measuring the performance of the last $200,000$ queries (or some other number in some cases). The costs are stated as the average number of blocks scanned *for each query that is not filtered out by result caching*, without list caching which will further improve performance. The baseline without projection caching is about $2,700$ block accesses per query. The overall performance across all three caching levels is evaluated in Section 6.4.

In our first experiment, we used the greedy algorithm from the previous section on a window of $250,000$ queries (the training window) directly preceding $250,000$ queries that were measured (the evaluation window). Thus, recent queries are analyzed by the greedy algorithm to allocate space in the cache for projections likely to be encountered in the future, and only these projections are allowed into the cache. There are two different ways in which this approach could be used: (1) After analyzing the queries in the training window, we could preload the projection cache with the projection selected by the greedy algorithm. This could be done say once a day during the night in a large bulk operation in order to improve performance during peak hours. (2) The second approach is to create the selected projections in an online fashion only when we encounter the corresponding pair in the evaluation window.

From Figure 7, we see that the performance of these two approaches is very similar across a range of cache sizes. The online method only benefits from

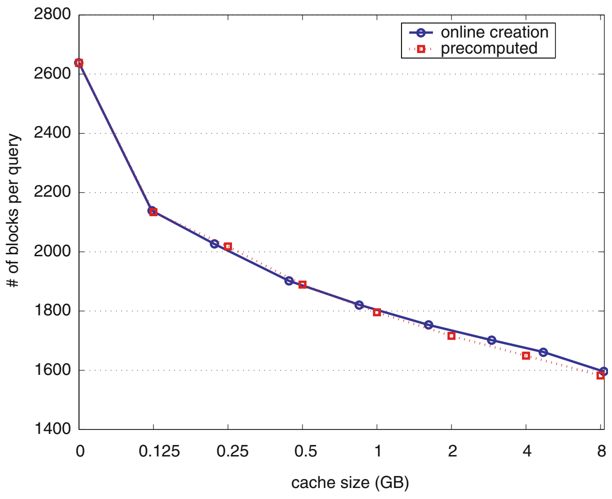


Figure 7 Block accesses per query for greedy projection caching with various cache sizes. Zero cache size means no projection caching. Shown are two curves, one for the case of precomputed projections and one where they are created online.

projection caching the second time a pair is encountered in the evaluation window, since the projection is created during the first time. The other method benefits even on the first occurrence, but uses almost half the cache space for precomputed projections that are never used in the evaluation window. Note that the cache size plotted in Figure 7 for the online variant is not the maximum cache size assumed by the greedy algorithm, but the amount of cache that is actually filled with projections (which is much lower than the size assumed by the greedy algorithm).

We observe that even with only 1 GB cache size (less than 10% of the index size), we already get a significant improvement to about $1,800$ blocks per query. In the online case, we did not include the overhead due to creation of intersections in Figure 7, but as shown in Table 1 the number of blocks written out per query is fairly small. For the other case, we ignore the cost of preloading the projections. The results indicate that there is probably not too much gained from precomputing projections, and that an online approach is preferable.

To avoid starting out with an empty cache at the start of each new window, we further optimized the online greedy algorithm by using a sliding window approach where periodically (e.g., every $100,000$ queries) we use the greedy algorithm to analyze a certain window of recent queries (say, $250,000$ queries). Any projection chosen by the greedy algorithm is marked as *protected*: once it is cached it cannot be evicted until it is unprotected in another run of the greedy algorithm. We also simultaneously keep cached projections from previous runs in cache as long as there is enough space in the cache. We experimented with various cache sizes and window sizes. The result in Figure 8 shows that this sliding window approach performs slightly worse than the simple greedy algorithm with online creation when the cache size is small, while it outperforms for larger caches. This is because in the sliding

Table 1 Cost of online projection creation in 4 KB block writes per query, for various amounts of cache space in GB.

Cache size	0.12	0.22	0.44	0.85	1.62	2.91	4.71
Blocks/query	0.13	0.24	0.47	0.89	1.70	3.05	4.94

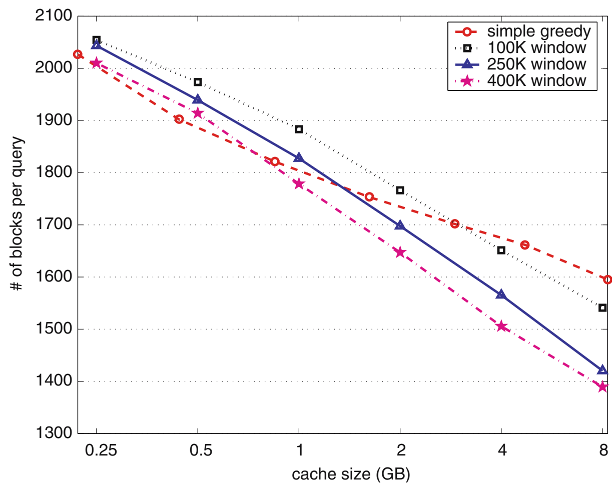


Figure 8 Block accesses per query for online greedy and for sliding-window greedy projection caching with various cache sizes and window sizes.

window algorithm, cached projections are inherited from the previous window, while online starts with an empty cache.

Performance of basic Landlord In Figure 9, we show results for the basic *Landlord* algorithm where we reset deadlines to their original values whenever a cached projection is used again. We observe similar performance in terms of the number of blocks scanned per query compared to the greedy algorithm. However, the amount of block writes in *Landlord*, shown in the second graph on top of the read costs, is quite high since a large number of projections are created and then quickly evicted from the cache without ever being used. Once we take this overhead into account, the basic *Landlord* approach does not provide any benefit for most cache sizes compared to not using projection caching at all.

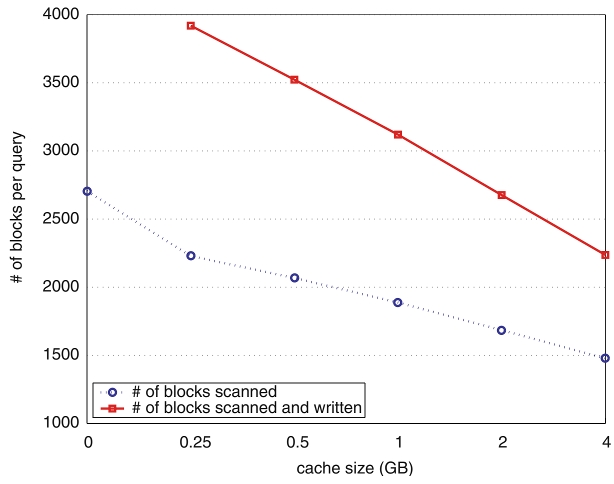


Figure 9 Block reads (lower graph) and block reads plus writes (upper graph) per executed query, without projection caching and with different amounts of cache space under the basic *Landlord* approach.

In the next subsection, we present several refinements of the basic *Landlord* approach that dramatically reduce the overheads of the approach while also further improving block read costs. The main idea is that we need an appropriate *cache admission policy*, in addition to a good cache eviction policy, to prevent unprofitable projections from being generated in the first place. We note that this is different from many other caching scenarios (e.g., web caching), where it may be preferable to just add all encountered objects to the cache and rely on the eviction policy to weed out useless entries.

6.1 Optimized Landlord policies

We now consider how to engineer the basic *Landlord* policy to improve performance. We present and evaluate two main ideas: (1) setting a more generous deadline for projections already in the cache that are being renewed, and (2) limiting the number of projections that are generated by not inserting every possible projection on the first encounter.

Landlord with α parameters In the basic *Landlord* approach a projection has its deadline reset to the original value whenever it is used again. In order to give a boost to projections that have already proved useful, versus newly encountered pairs of terms, we decide to give a longer deadline to projections that are being reset. (Thus, a tenant that renews gets a longer lease than a tenant that just moves in.) In particular, a renewed projection gets its original deadline plus a fraction α of its remaining deadline. In addition, we experimented with keeping a different fraction α' of the remaining deadline on the second and subsequent renewals. We experimented with a number of values for α and α' and found very good performance for $\alpha = 0.3$ and $\alpha' = 0.2$, though many other values between 0 and 1 achieve similar results.

Cache admission policies We experimented with several techniques for limiting the number of projections that are created and inserted into the cache. A simple approach is to never insert a projection for a pair of terms that has not been previously encountered (say, in the last few hours). A more refined rule would also take into account the cost of projection creation and the amount of benefit that results if the projection is used instead of the full lists.

After some experimentation we arrived at the following policy. We choose a window of the previous t queries, t to be determined later, for which we maintain statistics about the encountered queries. A projection $I_{a\to b}$ is only inserted into the cache if the corresponding pair of terms has *previously* occurred at least

$$\left\lceil \beta \cdot \frac{|I_{a\to b}|}{|I_a| - |I_{a\to b}|} \right\rceil$$

times within the last t queries, for some parameter β. (We looked at various values of β and found that values around 30 or higher seem to work best.) In particular, this means we never insert if the pair of terms has not previously occurred during these t queries. Moreover, decreasing t will result in fewer insertions into the cache, and a projection $I_{a \to b}$ that is not much smaller than the list I_a is only inserted if it has occurred repeatedly.

To apply this approach, we select an insertion overhead that we are willing to tolerate, say $b = 10$ blocks of projections that are written out per query on average. We start out with an initial value of t, say $t = 100,000$, periodically evaluate the average number of blocks written out per query, and then increase or decrease t to adjust for any discrepancy. We found that this converges quickly and results in a number of block writes per query very close to the target b. For example, for $b = 5$ and $\beta = 32$, we end up with a window of size around $300,000$ for a 4 GB cache.

The results of these optimizations are shown in Figure 10, which indicate that fine-tuning of the policies is extremely important in our scenario. The best approach based on the above rule performs fewer block reads than all other methods, and significantly less than any of the greedy methods discussed earlier, but is also quite conservative about inserting projections and thus minimizes write costs into the cache. (The precise cost of the best method was $1,603$ blocks for 1 GB and $1,253$ blocks for 4 GB.) In the next subsection, we show that this also results in small CPU overhead for piggybacked projection creation during query processing.

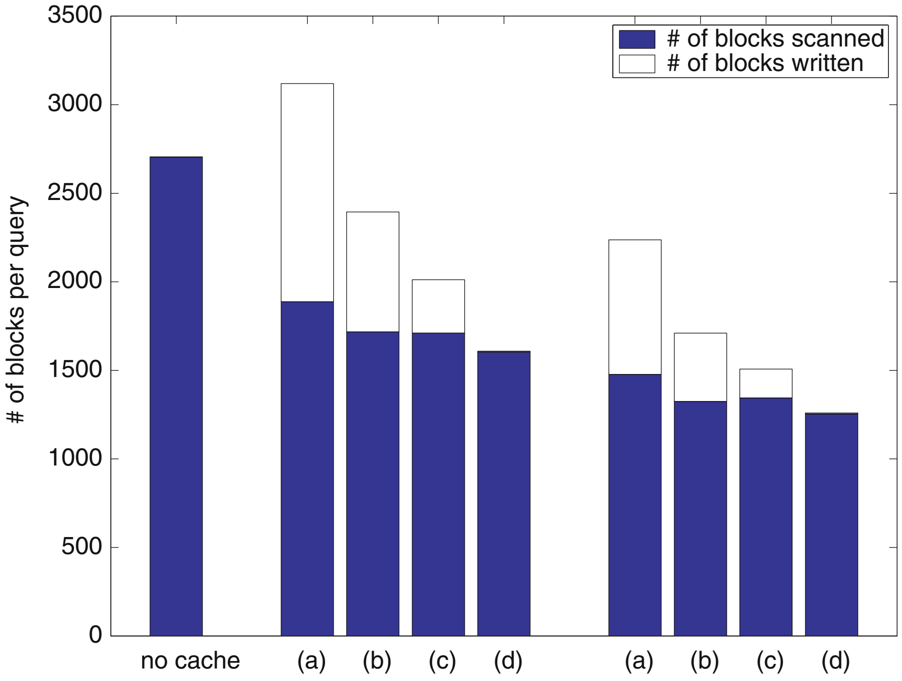


Figure 10 Comparison of no projection caching (*leftmost bar*), 1 GB cache size (*next* 4 *bars*), and 4 GB cache size (*rightmost* 4 *bars*), for various refinements of *Landlord*. For each cache size, we show the read cost (*solid portion of bar*) and write cost (*outlined*) for four policies from left to right: (a) basic *Landlord*, (b) with $\alpha = 0.3$ and $\alpha' = 0.2$, (c) same with insertion only on second encounter of a pair, and (d) with alphas and the cache admission policy described above. For the last case, we choose $b = 5$ and $\beta = 32$, resulting in negligible (in fact, invisible in this chart) write cost.

6.2 CPU overhead of creating projections

We now address the CPU overhead of creating projections in our query processor. To do so, we need to understand how the query processor generates intersections of inverted lists during normal query execution. This is done in a *document-at-a-time* manner by simultaneously scanning all lists. More precisely, we first scan an element from the shortest list, and then search forward for a matching element in the second-shortest list. If such an element is found, we search forward in the next list, otherwise we return to the shortest list. For queries with more than two or three keywords, it is quite common that many of the forward searches into the longest list can skip several blocks at a time, given an appropriate indexing scheme. Note that it is rare for skips to be long enough to improve disk performance on modern hard disks, since each inverted list is laid out sequentially on disk for optimized scanning. However, skipping blocks does result in significant savings in list decoding and processing time since we can avoid accessing the entire list (assuming a blocked coding scheme).

However, when generating a projection, say between the longest and the shortest list, we often have to decode almost all the blocks, resulting in higher CPU cost than normal query processing. The additional CPU cost is related to the size of the created projection, and thus policies that decrease the number of block writes for created projections also tend to do well in terms of CPU performance.

In Figure 11, we show the cost of query processing with optimized projection caching versus a query processor without projection caching, measured by executing queries on memory-resident inverted lists. We see that policies with low b, i.e., policies that are very conservative about generating projections, perform well in terms of CPU cost, and that CPU cost closely correlates with the total number of tuples encoded and decoded. Projection caching performs additional decoding and encoding of tuples during creation of projections, but saves on decoding later when the projections are used in other queries. Overall, we observe a 25% decrease in CPU cost, implying a 33% increase in query throughput in a CPU-limited system, while the benefit for disk-bound systems is even higher. We experimented with several block sizes for the blocked compression scheme, and observed similar relative

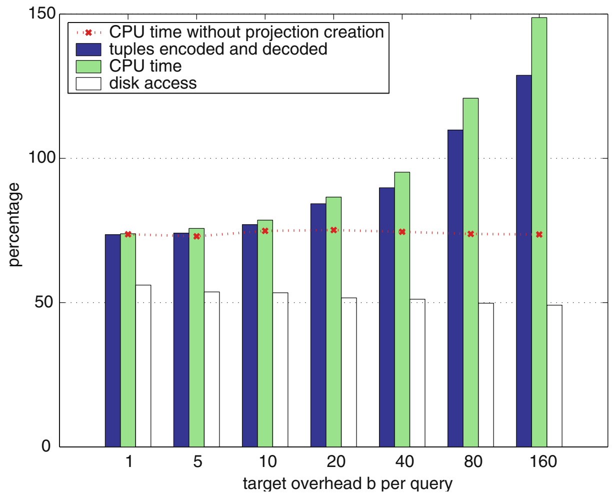


Figure 11 Cost per query for optimized *Landlord*, relative to a query processor without projection caching (100%). We show the number of tuples encoded and decoded, CPU time, and the number of disk blocks accessed. We also plot the CPU cost under the assumption that projection generation is free; for small values of the target overhead b this is very close to the total CPU cost. For larger values of b, this overhead is significant and total CPU cost is higher than in a system without projection caching.

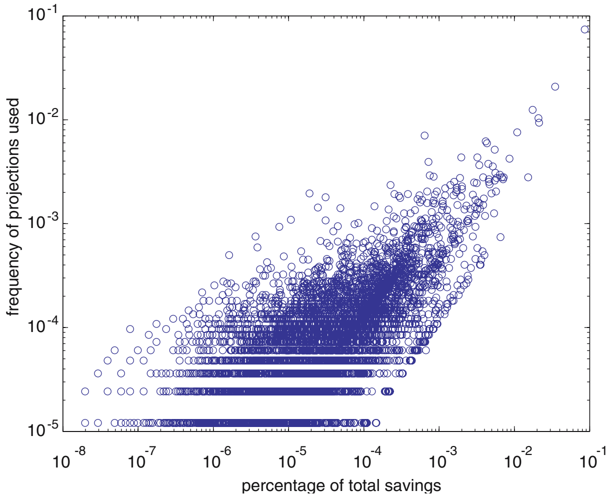


Figure 12 Scatter plot of total benefit versus frequency of use of the projections. Each circle represents a projection that was used at least once. The *x*-axis shows the total savings due to the projection, as a fraction of the total savings from all projections. The *y*-axis shows the frequency of that projection, as a fraction of the total frequency of all projections that have been used. (The distinctive bands of circles at the bottom contain projections that were used once, twice, etc.)

behavior from a few hundred bytes to several KB. (In absolute terms, the smaller block sizes result in lower CPU cost for query processing as they decode fewer postings overall, but the relative benefit of projection caching is about the same.)

In summary, the results show that projection generation can be done at fairly low total CPU and disk overhead during query execution. However, this does not mean that any projection can be created at almost no cost; instead the low total cost is achieved by carefully selecting a fairly small number of profitable projections that are then created.

6.3 Statistics of cached projections

We now look at some statistics on the generated projections in order to better understand the benefits of projection caching. In Figure 12, we first look at how savings

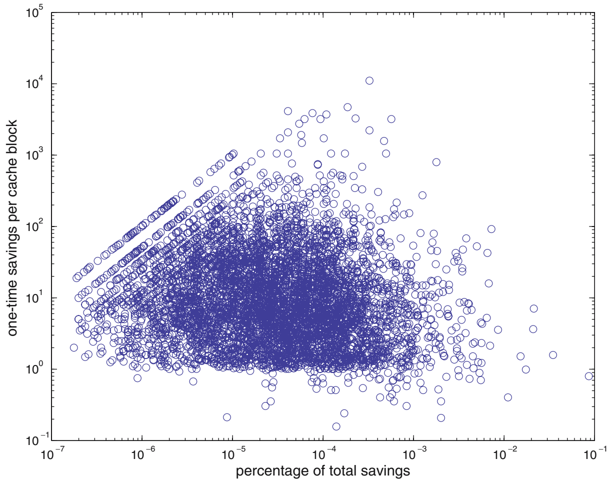


Figure 13 Total benefit due to a projection versus its benefit per cache block and per use.

from projections are related to their frequencies. Not surprisingly, a small number of very frequently used projections are responsible for a disproportionate share of the benefit. In particular, the top 100 most frequently used projections (corresponding to roughly the top half of Figure 12) account for almost 30% of total savings.

Next, we look at the total savings from a projection versus its benefit per cache block and per use, which is proportional to $\frac{|I_a|-|I_{a\to b}|}{|I_{a\to b}|}$. From Figure 13, we see that the projections from which most of the savings come actually have only relatively moderate per-use per-block benefit. We observed that these projections were also the ones taking up most of the cache space, and they had fairly long original lists. Thus, most of the benefit comes from projections involving reasonably common words, with long original lists, that are used a number of times, resulting in significant absolute savings rather than large relative savings.

As mentioned before, projection caching can be seen as related to the problem of building optimized index structures for phrase queries [6]. Thus, there is a question to what degree the utilized projections correspond to common phrases that could also be handled using a standard phrase index. By manually examining the generated projects, we found that this was generally not the case. While there were some projections that clearly corresponded to common phrases, such as $I_{new\to york}$ or $I_{high\to school}$, most of the projections and benefit were not of this type. Among these other projections, we noticed a fairly large number of projections such as $I_{can\to find}$, $I_{find\to pictures}$, or $I_{about\to learn}$, that seemed to originate from "question-type" queries. (The words in the projections were often not adjacent in the queries themselves.) Of course, it could be argued that some of these queries would be better handled through appropriate question answering techniques or at least some smart preprocessing.

6.4 Evaluation of multi-level caching

We now evaluate query processing performance over all three levels of caching. As suggested in [33] we use LRU for list caching. Inverted lists corresponding to projections are treated by the list caching mechanism just as any other inverted list (this turns out to perform best). For projection caching we use the optimized version

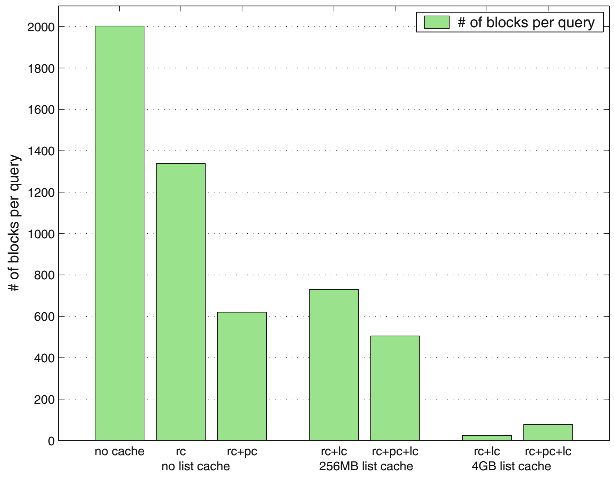
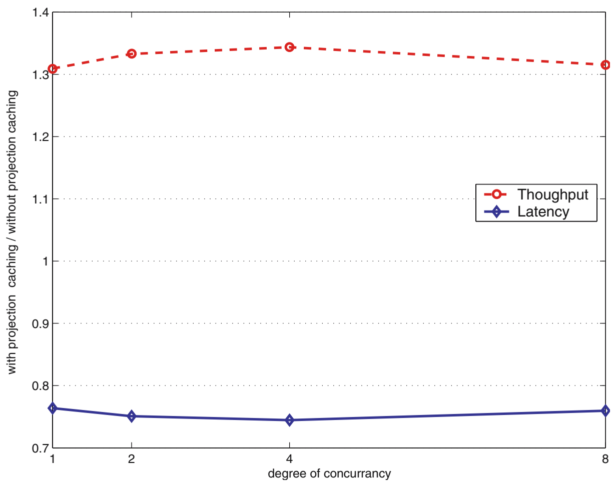


Figure 14 Number of block reads per query, for the following seven schemes (from left to right): no caching, result caching, result plus projection caching, result plus list caching and all three levels of caching with a 256 MB list cache, and result plus list caching and all three levels of caching with a 4 GB list cache.

Figure 15 Query throughput and latency in a query processor with projection caching, relative to a query processor without projection caching.



of *Landlord* from the previous subsection, with negligible overhead for generating projections online. Note that in the following, we report the average number of block reads over all queries, including those filtered out by result caching. This allows a comparison over all three levels of caching.

The results are shown in Figure 14. Without any caching, about 2, 000 blocks are read for each query; this is reduced to less than 1, 400 blocks using just result caching (with each surviving query actually having a cost of 2, 700 blocks as shown in Figure 7). This is brought down to about 620 blocks per query by adding projection caching. Using result caching and list caching with a 256 MB list cache, we get a performance of about 730 blocks, which is reduced to about 505 blocks by adding projection caching. Note that a cache of 256 MB is about 2.5% of the total index size of over 10 GB, and thus an example of a mainly disk-bound setup. On the other hand, when we have a list cache that can hold almost 40% of the total index (shown in the two rightmost bars), then disk access decreases to a very low level, which means that CPU becomes the primary bottleneck. In this case, projection caching increases disk accesses but reduces CPU work significantly as shown in the previous subsection, which is a desirable outcome for this case. (Essentially, if disk is not the bottleneck, it is better to fetch a small projection from disk than to use a much larger inverted list that is already in memory.)

Finally, we evaluate the actual improvement on query throughput and latency by using projection caching. We use a query processor developed in our lab with result caching and a 256MB list cache. We issued 2, 000 queries from our query log and measured the results with projection caching set off and on. As shown in Figure 15, query throughput is increased more than by 30%, while latency is decreased by around 25% when adding a 4 GB projection cache.

7 Concluding remarks

In this paper, we have proposed a new three-level caching architecture for web search engines that can improve query throughput. The architecture introduces a new intermediate caching level for search engines with AND query semantics (including essentially all current major engines) that can exploit redundancies in the query stream that are not captured by result and list caching in two-level architectures. Our experimental evaluation on a large query log from the Excite search engine showed significant improvements in performance due to the extra level of caching. However, actual performance is highly dependent on a good selection of caching policies and the system bottlenecks in the particular architecture.

There are several open questions that arise from this work. In particular, it would be interesting to perform a more formal study of the offline and online intersection caching problems defined in this paper. For example, one could study approximation results for the greedy heuristic, or competitive ratios for the *Landlord* approach in our scenario, or look at the case where we include the cost of generating projections into the corresponding weighted caching problem. Another interesting theoretical question concerns the performance of caching schemes on certain classes of input sequences, e.g., sequences that follow Zipf distributions on term frequencies.

It appears that the simple LRU scheme previously also used in [33] is actually not the best possible policy for list caching. In fact, we have recently seen interesting improvements based on adaptations of the *Landlord* algorithm with α and α' parameters to list caching. We note that this approach is also related to recent work by Megiddo and Modha [27] and others on caching policies that outperform LRU in a variety of applications. We are currently studying list caching policies in more detail.

On the more practical side, we expect that additional tuning of the caching policies and the availability of larger traces would show some additional gains, and we also plan to fully integrate intersection caching into our existing high-performance query processor. Another open question concerns the relationship between intersection caching and specialized index structures for common phrases.

Finally, it would be very interesting to evaluate combinations of caching and pruning techniques in future work. We believe that integrating projection caching into pruning techniques such as [25] should not be difficult for two reasons: First, as discussed a projection can be treated just as any other inverted list in the index. Second, we observed that under a good choice of policies and parameters, the overhead of generating projections is tiny, and would still be small even when aggressive pruning brings down the baseline cost. We note that many search engines appear to use the distance between the query terms in a page as an important factor in ranking. To our knowledge there is no published work on how to apply pruning to such types of ranking functions, which are not based on a simple combination of the scores for different terms.

References

1. Anh, V., Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. In: Proceedings of the 24th Annual SIGIR Conference on Research and Development in Information Retrieval, pp. 35–42, September 2001
2. Anh, V., Moffat, A.: Compressed inverted files with reduced decoding overheads. In: Proceedings of the 21st Annual SIGIR Conference on Research and Development in Information Retrieval, pp. 290–297, 1998

3. Arasu, A., Cho, J., Garcia-Molina, H., Raghavan, S.: Searching the web. ACM Trans. Internet Technol. **1**(1), June (2001)
4. Badue, C., Baeza-Yates, R., Ribeiro-Neto, B., Ziviani, N.: Distributed query processing using partitioned inverted files. In: Proceedings of the 9th String Processing and Information Retrieval Symposium (SPIRE), September 2002
5. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addision Wesley (1999)
6. Bahle, D., Williams, H., Zobel, J.: Efficient phrase querying with an auxiliary index. In: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 215–221, 2002
7. Bhattacharjee, B., Chawathe, S., Gopalakrishnan, V., Keleher, P., Silaghi, B.: Efficient peer-to-peer searches using result-caching. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems, 2003
8. Brewer, E.: Lessons from giant scale services. IEEE Internet Comput., pp. 46–55, August (2001)
9. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh World Wide Web Conference, 1998
10. Broder, A.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences, pp. 21–29, IEEE Comput. Soc. (1997)
11. Cao, P., Irani, S.: Cost-aware WWW proxy caching algorithms. In: USENIX Symposium on Internet Technologies and Systems (USITS), 1997
12. Chaudhuri, S., Gravano, L.: Optimizing queries over multimedia repositories. Data Eng. Bull. **19**(4), 45–52 (1996)
13. Demaine, E., Lopez-Ortiz, A., Munro, J.: Adaptive set intersections, unions, and differences. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 743–752, 2000
14. Fagin, R.: Combining fuzzy information from multiple systems. In: Proceedings of ACM Symposium on Principles of Database Systems, 1996
15. Fagin, R., Carmel, D., Cohen, D., Farchi, E., Herscovici, M., Maarek, Y., Soffer, A.: Static index pruning for information retrieval systems. In: Proceedings of the 24th Annual SIGIR Conference on Research and Development in Information Retrieval, pp. 43–50, September 2001
16. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proceedings of ACM Symposium on Principles of Database Systems, 2001
17. Garcia-Molina, H., Ullman, J., Widom, J.: Database System Implementation. Prentice Hall (2000)
18. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP Completeness. WH Freeman and Company (1979)
19. Haveliwala, T.: Topic-sensitive pagerank. In: Proceedings of the 11th International World Wide Web Conference, May 2002
20. Jonsson, B.T., Franklin, M.J., Srivastava, D.: Interaction of query evaluation and buffer management for information retrieval. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 118–129, June 1998
21. Kaszkiel, M., Zobel, J., Sacks-Davis, R.: Efficient passage ranking for document databases. ACM Trans. Inf. Sys. (TOIS) **17**(4), 406–439, October (1999)
22. Lempel , R., Moran, S.: Optimizing result prefetching in web search engines with segmented indices. In: Proceedings of the 28th International Conference on Very Large Data Bases, August 2002
23. Lempel, R., Moran, S.: Predictive caching and prefetching of query results in search engines. In: Proceedings of the 12th International World-Wide Web Conference, 2003
24. Li, J., Loo, B., Hellerstein, J., Kaashoek, F., Karger, D., Morris, R.: On the feasibility of peer-to-peer web indexing. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems, 2003
25. Long, X., Suel, T.: Optimized query execution in large search engines with global page ordering. In: Proceedings of the 29th International Conference on Very Large Data Bases, September 2003
26. Markatos, E.: On caching search engine query results. In: 5th International Web Caching and Content Delivery Workshop, May 2000
27. Megiddo, N., Modha, D.: Outperforming LRU with an adaptive replacement cache. IEEE Comput., pp. 58–65, April (2004)
28. Melnik, S., Raghavan, S., Yang, B., Garcia-Molina, H.: Building a distributed full-text index for the web. In: Proceedings of the 10th International World Wide Web Conference, May 2000

29. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. J. Am. Soc. Inf. Sci., **47**(10), 749–764, May (1996)
30. Richardson, M., Domingos, P.: The intelligent surfer: Probabilistic combination of link and content information in pagerank. In: Advances in Neural Information Processing Systems, 2002
31. Risvik, K., Aasheim, Y., Lidal, M.: Multi-tier architecture for web search engines. In: First Latin American Web Congress, pp. 132–143, 2003
32. Risvik, K., Michelsen, R.: Search engines and web dynamics. Comput. Netw. **39**, 289–302 (2002)
33. Saraiva, P., de Moura, E., Ziviani, N., Meira, W., Fonseca, R., Ribeiro-Neto, B.: Rank-preserving two-level caching for scalable search engines. In: Proceedings of the 24th Annual SIGIR Conference on Research and Development in Information Retrieval, pp. 51–58, September 2001
34. Scholer, F., Williams, H., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proceedings of the 25th Annual SIGIR Conference on Research and Development in Information Retrieval, pp. 222–229, 2002
35. Shkapenyuk, V., Suel, T.: Design and implementation of a high-performance distributed web crawler. In: Proceedings of the International Conference on Data Engineering, 2002
36. Suel, T., Mathur, C., Wu, J., Zhang, J., Delis, A., Kharrazi, M., Long, X., Shanmugasundaram, K.: ODISSEA: A peer-to-peer architecture for scalable web search and information retrieval. In: International Workshop on the Web and Databases (WebDB), June 2003
37. Tomasic, A., Garcia-Molina, H.: Performance of inverted indices in distributed text document retrieval systems. In: Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS), 1993
38. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, second edition (1999)
39. Xie, Y., O'Hallaron, D.: Locality in search engine queries and its implications for caching. In: IEEE Infocom 2002, pp. 1238–1247, 2002
40. Young, N.: On-line file caching. In: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 82–86, 1998