Cost-aware Intersection Caching and Processing Strategies for In-memory Inverted Indexes

Esteban Feuerstein  
Dept. de Computación, FCEyN-UBA  
Buenos Aires, Argentina  
efeuerst@dc.uba.ar

Gabriel Tolosa  
Dept. de Cs.Básicas, UNLu  
Dept. de Computación, FCEyN-UBA  
Buenos Aires, Argentina  
tolosoft@unlu.edu.ar

ABSTRACT

We propose and experimentally evaluate several combinations of cost-aware intersection caching policies and evaluation strategies for intersection queries for in-memory inverted indexes. We show that some of the combinations can lead to significant improvements in the performance at the search-node level. Dynamic policies are better than the others achieving on average a 30% time reduction compared with the best hybrid policy.

Keywords  
search engine, intersection caching, processing strategies

1. INTRODUCTION

Modern general purpose search engines (SE) receive millions of queries per day that must be answered under strict performance constraints, namely in not more than a few hundred milliseconds [23, 21]. According to [13], optimizing the efficiency in a commercial SE is also important because this translates into financial savings for the company.

Usually, the architecture of a SE is formed by a front-end node called broker and a large number of machines [2] (search nodes) organized in a cluster that process queries in parallel. Each search node holds only a fraction of the document collection and maintains a local inverted index that is used to obtain a high query throughput. Given a cluster of p search nodes and a collection of C documents and assuming that these are evenly distributed among the nodes, each one maintains an index with information related to only \( \frac{C}{p} \) documents. Besides, to achieve the strong performance requirements, SE usually implement different optimization techniques such as posting list compression [27] and pruning [16], results prefetching [13] and caching [1].

Caching can be implemented at several levels. At broker level, a results cache [18] is generally maintained, which stores the final list of results corresponding to the most frequent or recent queries. This type of cache has been extensively studied in the last years [19], as it produces the best performance gains, though achieving lower hit ratios compared with the other caching levels. This is due to the fact that each hit at this level implies very high savings in computation and communication time. The lowest level is posting list caching which simply maintains a memory buffer to store the posting lists of some popular terms. This cache achieves the highest hit ratio because query terms appear individually in different queries more often than their combinations. Complementarily, an intersection cache [15] may be implemented to obtain additional performance gains. This cache attempts to exploit frequently occurring pairs of terms by keeping in the memory of the search node the results of intersecting the corresponding inverted lists. Figure 1 shows the architecture of a SE including the basic caching levels.

The idea of caching posting-list intersections arises rather naturally in this framework. However, the problem of combining different strategies to solve the queries considering the cost of computing the intersections is a rather challenging issue. Furthermore, we know that the number of terms in the collection is finite (although big) but the number of potential intersections is virtually infinite and depends on the submitted queries. Also, some high-scale search engines actually maintain the whole inverted index in main memory. In this case, the caching of lists becomes useless, but an intersection cache may save processing time.

The two main query evaluation strategies are Term-at-a-time (TAAT) and Document-at-a-time (DAAT) [24]. With...
implement an intersection cache policies which take into account the cost of the queries to execute, that are beyond our scope but need to be addressed in future work. Experimental results suggest that the most efficient way to solve a conjunctive query is to intersect the two shortest lists first, then the result with the third, and so on, as in a TAAT strategy [14].

For each type of cache, a number of item replacement policies have been developed and tested to achieve the best overall system performance. One difference among the different caching levels is that in some of them the size of the items is uniform (as for example in the results cache) while in the others item sizes may vary (lists or intersections caching). This implies that the value or benefit of having an item in cache depends not only on its utility (i.e. the probability or frequency of a hit) but also on the space it occupies. Moreover, there is another parameter that may be taken into account, that is the saving obtained by a hit on an item (i.e. how much time the computation of the item would require).

In this work we are concerned with this kind of problem with the aim of reducing the query execution time. We propose and evaluate the performance of cost-aware eviction policies for the intersections cache when the full inverted index resides in main memory. This article extends a previous work [10] where we evaluate the intersection caching problem for disk indexes. Here, we also propose and test a different way of combining the query terms to obtain more (and more valuable) cache hits using both static, dynamic and hybrid caching policies. Our approach assumes TAAT conjunctive query processing that enables the proposed strategies.

1.1 Our contributions

In this paper, we carefully analyze and evaluate caching policies which take into account the cost of the queries to implement an intersection cache. We implement static, dynamic and hybrid policies adapting to our domain some cost-aware policies from other domains. The various caching policies can be combined with different query-evaluation strategies, some of which are general-purpose but also one originally designed to take benefit of the existence of an intersection cache by reordering the query terms in a particular way. Finally, we experimentally evaluate several combinations of the caching policies and evaluation strategies when the whole inverted index fits in main memory. Our evaluation is done using a real web crawl dataset and a well-known query log.

The remainder of the paper is organized as follows: The introductory part is completed in Subsection 1.2, where we present related work. In Section 2 we explain different query evaluation strategies (including a new one). Section 3 is devoted to introduce the different policies while sections 4 and 5 present the setup and results of our experiments. Finally, Section 6 presents conclusions and future research directions.

1.2 Related Work

Caching techniques in the context of web search engines have become an important and popular research topic in recent years. Much work has been developed in issues such as lists caching and results caching but intersection caching has not received enough attention.

Result caching consists in keeping a cache of recently returned results at the front machine to speedup identical queries repeatedly issued. Markatos [18] shows that there exists a temporal property in query requests analyzing a web search engine query log. That work compares static vs dynamic policies showing that the former perform better for caches of small sizes while the latter are advantageous for medium-sizes caches.

The work in [7] considers hybrid policies (SDC) for populating the cache, that is, a combination of a static part for popular queries over the time and dynamic part for dealing with bursty queries. Gan and Suel [11] consider the weighted result caching problem instead of focusing on hit ratio, while [19] propose new cost-aware caching strategies and evaluate these for static, dynamic and hybrid cases using simulation. The caching of posting lists is a well known approach to reduce the processing cost of queries that contain frequent terms. Baeza-Yates [1] studied the trade-off between result caching and list caching. They show that posting lists caching achieves a better hit rate because the repetitions of terms are more frequent than repetition of queries but they do not use processing cost as an optimization target. In [27], the performance of compressed inverted lists for caching is studied. The authors compare several compression algorithms and combine them with caching techniques, showing how to select the best setting depending on two parameters (disk speed and cache size). Using another approach, Skobeltsyn [22] combined pruned indexes and result caching.

The intersections cache is first used in [15] where a three-level approach is proposed for efficient query processing. In this architecture a certain amount of extra disk space (20-40%) is reserved for caching intersections. They also say that the use of a Landlord policy improves the performance. This idea is also taken in [5] where results of frequent subqueries are cached and a similar approach is exploited in [6] for speeding up query processing in batch mode.

Finally, [17] and [9] consider intersection caching as part of their architecture. In the former work, the authors propose a five level cache hierarchy combining caching at broker and search node level while the latter presents a methodology for predicting the cost of different architectures for distributed indexes. In [8], a three-dimensional architecture that includes an intersection cache is also introduced.

2. QUERY PROCESSING

In this section we provide basic background about the execution of a query in a distributed search engine. A query \( q = \{t_1, t_2, \ldots, t_n\} \) is a set of terms that represents the user’s information need, as the intersection \( \cap_{i=1}^{n} t_i \). When the SE receives a query, its processing is split into two phases: (a) the retrieval of the data from the index (in memory or disk) and computation of the partial answer, and (b) the production of the response page to be sent to the user [3].

In a first phase, the broker machine receives the queries and tests first the result cache looking for a match. If no hit is found, the query is sent to the backend servers (search
nodes) in the cluster where an inverted index resides. This data structure contains the set of all unique terms in the document collection (vocabulary) associated to a posting list that contains the documents assigned to that node where this term occurs. Actually, the posting list contains docID of these documents and additional information used for ranking (e.g. the within-document frequency of this term occurs. Actually, the posting list contains \( t_d \)). For more details about index data structures, please refer to [25].

Each search node searches for the posting lists of the query terms, executes the intersection of the sets of documents IDs and finally ranks the resulting set. After that, each search node sends the list of top-\( k \) documents to the broker which merge these to get the final answer. This is a time consuming task that is critically important for the scalability of SE so different cache levels are used to improve performance.

A second phase of the computational process consists typically of taking, at the broker level, the top ranked answers to make the final result page. This is composed of titles, snippets and URL for each resulting item. As the result page is typically made of 10 documents the cost of the processing may be considered constant as it is only slightly affected by the size of the document collection. We focus our attention on the first phase of the query processing task by using the intersection cache and different strategies to solve the query.

### 2.1 Strategies for intersecting posting lists

When a search node receives a query \( q = \{t_1, t_2, t_3, \ldots t_n\} \), it is natural to reorder the terms \( t_1 \ldots t_n \) in ascending order of their posting lists lengths, and compute the resulting list \( R \) according to this basic strategy (S1):

\[
R = \bigcap_{i=1}^{n} t_i = (((t_1 \cap t_2) \cap t_3) \ldots \cap t_n)
\]

This aims at compute the pairwise intersection of the smallest possible lists. For example, the four-term query \( q = \{t_1, t_2, t_3, t_4\} \) would be computed as \(((t_1 \cap t_2) \cap t_3) \cap t_4\).

In presence of an intersection cache other strategies can be implemented. We propose two more methods. The first (S2) splits the query in pairs of terms, as:

\[
R = \bigcap_{k=1}^{n/2} (t_{2k-1} \cap t_{2k}) = ((t_1 \cap t_2) \cap (t_3 \cap t_4) \ldots \cap (t_{n-1} \cap t_n))
\]

In the case of the previous four-terms query, the evaluation order becomes: \((t_1 \cap t_2) \cap (t_3 \cap t_4)\). The other strategy (S3) uses two-term intersections but overlaps the first term of the \( i \)-th intersection with the second term of the \((i-1)\)-st intersection as:

\[
R = \bigcap_{k=1}^{n/2} (t_{2k-1} \cap t_{2k+1}) = ((t_1 \cap t_2) \cap (t_2 \cap t_3) \ldots \cap (t_{n-1} \cap t_n))
\]

resulting in: \((t_1 \cap t_2) \cap (t_2 \cap t_3) \cap (t_3 \cap t_4)\) for the example.

Using S1 the only chance of obtaining a hit in the intersection cache is to find \((t_1 \cap t_2)\) while S2 and S3 combine more terms in pairs that are tested against the cache. For example, using S2 in a four-term query we will look in the cache for \((t_1 \cap t_2)\) and \((t_3 \cap t_4)\). The obvious potential drawback is that S2 and S3 may require higher cost to compute more (S3) or more complex (S2) intersections. However, this additional cost may be amortized by the gain obtained through more cache hits.

### 2.2 A new strategy using cached information

As we show before, strategy S1 only tests for one candidate intersection while S2 and S3 test for two and three intersections respectively. As we have just mentioned that redundancy may be positive in presence of intersection cache, we consider this fact and give an additional step by developing a new strategy that first tests all the possible two-term combinations in the cache in order to maximize the chance of getting a cache hit. We call this strategy S4.2.

Let’s consider the following example: Given \( q = \{t_1, t_2, t_3, t_4\} \) and \((t_2 \cap t_3)\) and \((t_1 \cap t_4)\) are cached intersections we set \( A = (t_2 \cap t_3) \) and \( B = (t_1 \cap t_4)\) and rewrite the query as \((A \cap B)\cap t_1\) which only needs to manage the posting list of \( t_1 \). Notice that each time a query is evaluated, we need to check \( \binom{n}{2} \) candidate pairs. This number is obviously a function of the maximum length of a query. However, the distribution of the number of terms in a query shows that 95% of queries have up to 5 terms. Despite this observation, we measured the computational cost overhead incurred doing this (across all the queries in the log) and we found that on average the increment of time is about 0.13%.

Finally, we considered a related policy (S4.2), that allows caching and a set of items with associated sizes and costs. The number of items that fit in the cache is related to their sizes so the challenge is to maintain in the cache the most valuable ones.

In most cases, we consider only two-term intersections to insert in cache. In the case of S4 we also introduce a variation with three-terms intersections (S4.3). We use the following notation: \( F_i \) is the frequency of the intersection \( I \) (computed from a training set); \( c_I \) is the cost of the intersection \( I \) (calculated from a real scenario); \( S_I \) is the size of the resulting list (i.e. \(|t_1 \cap t_2|\) and \( k \) is a scaling factor used to enhance the importance of \( F_i \), as used in [19]).

### 3. INTERSECTION CACHING POLICIES

In the previous section we have explained how the list intersections are computed. We will now describe different caching policies that can be used to decide which intersections are kept in the cache. The task of filling the cache with a set of items that maximize the benefit in the minimum space can be seen as a version of the well-known knapsack problem, as discussed in previous work related to list caching [1]. Here, we have a limited predefined space of memory for caching and a set of items with associated sizes and costs. The number of items that fit in the cache is related to their sizes so the challenge is to maintain in the cache the most valuable ones.

In most cases, we consider only two-term intersections to insert in cache. In the case of S4 we also introduce a variation with three-terms intersections (S4.3). We use the following notation: \( F_i \) is the frequency of the intersection \( I \) (computed from a training set); \( c_I \) is the cost of the intersection \( I \) (calculated from a real scenario); \( S_I \) is the size of the resulting list (i.e. \(|t_1 \cap t_2|\) and \( k \) is a scaling factor used to enhance the importance of \( F_i \), as used in [19]).

#### 3.1 Static Policies

Static caching is a well-known approach used both in list caching and in result caching. It consists on filling a static cache of fixed capacity with a set of predefined items that will not be changed at run-time. The idea is to store the most valuable items to be used later during execution. We evaluated different metrics for filling the cache:

- **FB (Freq-Based):** Fills the cache with the most frequent intersections.
- **CB (Cost-Based):** Fills the cache with the most costly intersections.
- **FC (Freq*Cost):** Uses the items that maximize the product \( F_i \times c_i \).
- **FS (Freq/Size):** Computes the ratio \( \frac{F_i}{S_i} \) and chooses the items that maximize that value. It is similar to \( Q_T:F_T \times C_T \) [1]
used to fill a static list cache. There “value” corresponds to $f_s(t)$ and “size” correspond to $f_d(t)$.

- **FkC (Freq**$^*$Cost): Like FC but prioritizes higher frequencies according to the skew of the distribution [19].
- **FCS (Freq**$^*$Cost/Size): This is a combination of the three features that define an intersection trying to best capture the value of maintaining an item in the cache.
- **FkCS (Freq**$^*$Cost/Size): Following the previous idea, but emphasizing $F_t$.

### 3.2 Dynamic Policies

Static caching is an effective approach to exploit long-term popularity of items but it can not deal with bursts produced in short intervals of time where a dynamic approach handles this case in a better way. A dynamic policy does not require of a training set of data to compute the “best” candidates to be inserted in cache but requires a replacement policy to decide which item evict when the cache is full.

- **LFU**: This strategy maintains a frequency counter for every item in the cache, which is updated when a hit occurs. The item with the lowest frequency value is evicted when space is needed.
- **LFUw**: This is the online (dynamic) version of the FC static policy. The score is computed as $F_t \times C_t$. It is called LFUw in [11]
- **LRU**: Chooses for eviction the least recently referenced item when a hit occurs. The item with the lowest frequency value is evicted when space is needed.
- **LCS**: Introduced in [19] for result caching. Each cached item has an associated cost (computed according the appropriate model). When a new item appears, the least cost cached element is evicted. This is the online version of CB.
- **FCSol**: Online version of the static FCS policy. The value $F_t \times \frac{\text{Cost}}{\text{Size}}$ is computed at run-time for each item.
- **Landlord**: Whenever an item is inserted in the cache a credit value is assigned (proportionally to its cost). When the cache is full, the item with the smallest credit is selected for eviction. At that moment, the credit value of all the remaining objects is recalculated deducting the value of the evicted item from their credit [26].
- **GDS**: GreedyDual-Size [4]. For each item in the cache, it maintains a value $H_{\text{value}} = \frac{\text{Cost}}{\text{Size}} + L$. L acts as an aging factor. The cached item with the smallest H-value is selected for eviction and its $H_{\text{value}}$ is set into L. During an update, the $H_{\text{value}}$ of the requested item is recalculated since L might have changed.

### 3.3 Hybrid Policies

Hybrid caching policies make use of two different sets of cache entries arranged in two levels. The first level contains a static set of entries which is filled on the basis of usage data. The second level contains a dynamic set managed by a given replacement policy. When an item is requested, it looks first into the static set and then into the dynamic set. If a cache miss occurs the new item is inserted into the dynamic part (and may trigger the eviction of some elements). The idea behind this approach is to manage frequent items with the static part and recent items with the dynamic part.

- **SDC**: This stands for “Static and Dynamic Cache” [7]. The static part of SDC is filled with the most frequent items while the dynamic part is managed with LRU cache.
- **FCS-LRU**: Variant of SDC where the static part is filled with the items with the highest $F_t \times C_t$ value.
- **FCS-Landlord**: Similar to the previous policy but the dynamic part is Landlord.
- **FCS-GDS**: In this case, the dynamic part is managed by the GDS policy.

### 4. SETUP AND METHODOLOGY

Our aim was to emulate a search node in a search engine cluster. According to [2] this kind of cluster is made up of more than 15,000 commodity class PCs. In our experiments, we simulate one of the nodes of a search engine running in a cluster of that size using an Intel(R) Core(TM)2 Quad CPU processor running at 2.83 GHz and 8 GB of main memory.

**Document collection**: We use a subset of a large web crawl of the UK obtained by Yahoo! It includes 1.479.139 documents, with 6.493.453 distinct index terms and takes 29 GB of disk space in HTML format. We use Zettair\(^1\) search engine to index the collection achieving a final index size of about 2.17 GB in compressed format (Zettair uses a variable-byte compression scheme for posting lists with a B+Tree vocabulary structure). Each entry in the posting lists includes both the docID and their term frequency (used for ranking).

**Query log**: We use the well known AOL Query Log\(^2\) [20] which contains around 20 million queries. We divide the query log in three parts. The first is used to compute some statistics for filling static caches. The second, is used to warm-up the cache and the remaining are reserved as the test set (3.9 million queries). All queries are normalized using standard techniques.

**Cost Model**: We model the cost of processing a query in a search node in terms of CPU times ($C_{cpu}$) because we assume that the full index fits in main memory. The total cost of executing a query depends on the number of intersections that resides in cache (i.e. cache hits) and the time consumed to intersect the resulting lists. To calculate $C_{cpu}$ we run a list intersection benchmark. We omit the details due to space constraints.

### 5. EXPERIMENTS AND RESULTS

In this section, we provide a simulation-based evaluation of the proposed strategies and caching policies. The total amount of memory reserved for the intersection cache ranges from 1 to 16GB (that can hold about 40% of all possible intersections of our dataset). For each query (and each intersection strategy), we log the total cost incurred using the cache in each case. We normalize the final costs to allow a more clear comparison among strategies and policies (figures are in comparable scales).

#### 5.1 IC with static policies

We use the seven static caching policies and the four proposed strategies (S1..S4). As we expected, the best strategy is S4. Abusing notation, we find that the ordering of the remaining policies is (S3 > S1 > S2) for big caches and S4 is 16% better (on average) than S2. The most competitive policy is FS (Figure 2). As we mentioned earlier, it is known that FS achieves the best hit rate for list caching [1], which leads to the best performance on intersection caching when the costs of the items are small.

---

2. Available at http://imdc.datcat.org/collection/1-003M-5
Figure 2: Performance of static policies. Strategies S1..S3 (top) and S4 (bottom)

We observe that the gains of computing extra intersections in S3 (to improve the hit rate) is not compensated with the saving due to cache hits because the involved costs are small (only CPU time). Another observation is the poor performance of CB and the little improvement when increasing the cache size. The most costly items are not frequent enough so this policy performs badly because the gains of a cache hit are small. Looking in depth only S4, we see that the three policies perform similarly (differences around 1%).

5.2 IC with dynamic policies

In this section we report the results of pure dynamic policies. Our new strategy S4 is the best and the global ordering of the remaining ones is the same as in the previous section (S3 > S1 > S2). Again, cost-based replacement criterion (LCU) performs badly in all cases. Also, we find that LRU is the best policy for strategies S1, S2 and S3 (Figure 3) where the best cost-aware policy (Landlord) performs similarly (except for the smallest cache size). We extend the study of S4 incorporating another cost-aware policy (GDS) considering two-term (S4I2) and three-term (S4I3) intersections. In general, S4 is 30% better than S2 (on average). The GDS policy clearly outperforms LRU and Landlord (28% better on average). Even more, we get an improvement of around 12%, 9% and 3% (respectively against LRU, Landlord and GDS) when using three-term intersections.

5.3 IC with hybrid policies

Finally, we evaluate hybrid caching policies mentioned in section 3.3. We observe that it is possible to obtain improvements also when the dynamic portion of the cache is cost-aware. We find similar relation among the strategies (S3 > S1 > S2 > S4) as in previous cases. Figure 4 shows the results. Using SDC as a baseline we got an improvement close to 7% with FCS-GDS for S1 while Landlord is about 3% better for large cache sizes.

In the case of S4, we test four combination. We used FCS for the static part because it was competitive in the previous cases and compare Landlord vs GDS. Here, we also use two-term and three-term intersections. In both cases, we achieve up to 8% reduction in the total cost using GDS for cache sizes of 1, 2 and 4GB while there are no differences with bigger caches. Comparing both approaches, incorporating three-term intersections leads to an improvement close to 3% (on average).

6. CONCLUSIONS AND FUTURE WORK

We propose and experimentally evaluate several combinations of cost-aware intersection caching policies and evaluation strategies for in-memory inverted indexes. The main result is that in all cases, our new evaluation strategy (S4) outperforms the others regardless the type of policy. Also, that cost-aware replacement algorithms are better than cost-oblivious ones. The overall results here show that dynamic policies are better than the others with GDS as the best replacement strategy achieving on average a 30% time reduction compared with the best hybrid policy. In the case of static caching, FS is the best choice. As it was pointed
out, FS is best for hit ratio which leads to a better performance too when the costs are smaller.

As future work, we plan to design and evaluate specific data structures for cost-aware intersection caching. The optimal combination of terms to intersect and maintains in cache seems to be still an open problem, especially for cost-aware approaches. Also, it would be interesting to evaluate combinations of other techniques such as list pruning and compression to this case because under the scenario where the full inverted index fits in main memory the intersection caching becomes the second level cache in a search node.

7. ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their detailed and valuable comments and suggestions to improve the final version of the paper.

8. REFERENCES

Upper-bound approximations for dynamic pruning. 


