A Data Structure for Sponsored Search

Arnd Christian König, Kenneth Church, Martin Markov

Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
{chrisko, church, mmarkov}@microsoft.com

***Abstract*— Inverted files have been very successful for document retrieval, but sponsored search is different. Inverted files are designed to find documents that match the query (all the terms in the query need to be in the document, but not vice versa). For sponsored search, ads are associated with bids. When a user issues a search query, bids are typically matched to the query using *broad-match* semantics: all the terms in the bid need to be in the query (but not vice versa). This means that the roles of the query and the bid/document are reversed in sponsored search, in turn making standard retrieval techniques based on inverted indexes ill-suited for sponsored search. This paper proposes novel index structures and query processing algorithms for sponsored search. We evaluate these structures using a real corpus of 180 million advertisements.**

I. Introduction

Sponsored search is different from web search and document retrieval. In web search and document retrieval, the algorithm is the challenge. The researcher cannot change the document collection, which is fixed in advance, but the researcher is encouraged to propose a new ranking algorithm that improves recall and precision. Sponsored search is different. Bidding is the challenge. The advertiser cannot change the ranking algorithm, which is fixed in advance, but is encouraged to bid the right amount on the right phrases to reach the desired target audience within certain budget constraints.

Sponsored search uses a number of common matching algorithms, with the most important being *broad match*. Broad match requires all the words in the bid phrase to be in the query, but not vice versa. Thus, the bid "*used books*", for example, matches the query *cheap used books*, but not the queries, *books* or *comic books*. Once all matching ads have been retrieved, additional filters are applied which make use of a number of secondary criteria: bid price, keyword-exclusion, clicked-through rate, overlap with advertisements displayed earlier.... The ads that win the auction are then ranked and displayed.

Broad match is the default matching algorithm in sponsored search. While there are other matching algorithms such as *exact match* and *phrase match*, the vast majority of advertisements use broad match (in the real-life advertisement corpora we studied, over 90% of all advertisements enabled broad-matching). The retrieval problem underlying broad match is very different from information retrieval for search. The key operation in information retrieval is the processing of *containment* queries. In these, the retrieval task is to return documents containing a *superset* of the keywords occurring in a query from an indexed corpus. In broad-match queries, the roles of query and corpus are reversed: the indexed corpus consists of phrases associated with individual advertisements and the retrieval task is to retrieve all advertisements whose phrases contain a *subset* of the keywords in the search query.

A. Opportunities for Improvement

This – in turn – means that the data structures used elsewhere in IR, e.g., inverted indexes indexing every word in an advertisement phrase, are not efficient when answering broad-match queries. To illustrate this, first consider the use of inverted indexes containing advertisement IDs as postings. Using them, it is possible to obtain the set of all advertisements whose phrases *overlap* with the query by forming the union of the postings in the inverted indexes corresponding to keywords in the query. However, at this point it is necessary to filter out advertisements whose bid phrase contains words not in the query, which is an operation that is not directly supported by inverted indexes, but requires either supplemental data structures or accessing the corresponding bids themselves.

One small modification that enables broad-match processing using inverted indexes only is to store the total number of keywords in the corresponding bid phrase together with each posting. However, the resulting access methods still require very large lists of postings to be processed for queries that contain at least one keyword that is frequent in the corpus of bid phrases, most of which typically are not matches of the query. We will analyze this phenomenon together with the described algorithms and their performance in detail later in the paper. To overcome this, we want to propose a data structure and query processing algorithm optimized for broad-match retrieval, which ideally processes a much smaller set of *candidate bids* only. For this, we will first describe the relevant properties of the data distribution found in large advertisement corpora, which our approach leverages.

B. Properties of Advertisement Corpora

The data distribution underlying the broad match retrieval task is very different from the ones encountered in traditional information retrieval. The indexed phrases are typically very short, with their word-length distribution close to the word-length distribution of queries itself. An example distribution from a corpus of 290 million real advertisements is displayed in Figure 1. Note that the Y-axis is in logarithmic scale, meaning that the drop-off from the highest point at 3 keywords

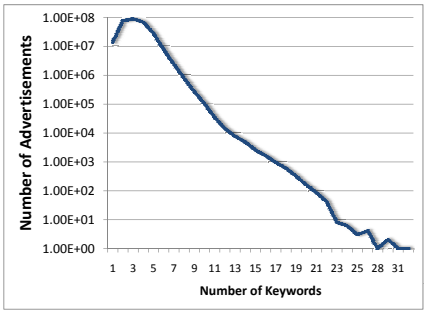

Fig. 1. Bids are short. In a corpus of 290 million advertisements, most bids (99.8%) are 8 words or less.

is quite rapid: 62% of all phrases have 3 keywords or less, 96% of them 5 keywords or less and 99.8% 8 keywords or less.

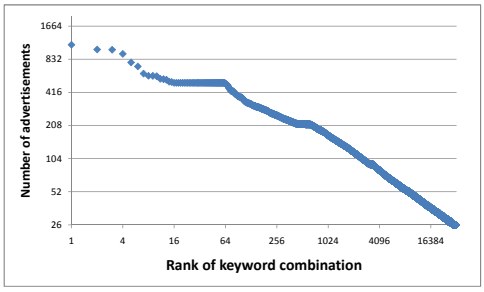

Fig. 2. The number of advertisements for a word-set obeys a Long Tail (Zipf Law) distribution. This plot is based on a corpus of 1.8 million advertisements.

Interestingly, the number of advertisements for a word-set (i.e. the set of words in each bid) obeys a Long Tail (Zipf Law) distribution, which means that most sets of words are associated with a small number of advertisements, even when the phrases are short. To illustrate this, we plotted the frequency distribution for the top 32K most frequent word-combinations in the advertisement phrases in a corpus of 1.8 million advertisements in Figure 2. This has the effect that the numbers of advertisements retrieved by broad-match processing (and from which the ads to display are are subsequently chosen) are small for virtually all queries. Therefore, pushing any of the secondary criteria used to rank/exclude advertisements into the index (similar how partial scores/impacts are used for early termination in traditional IR (e.g., [1], [2])) is less likely to result in noticeable performance improvement for ad retrieval.

Moreover, the eventual ranking of advertisements may take into account a number of different factors (e.g., the observed click-through rate for a particular advertisement, matching terms between bid and query, etc.), some of which are independent of the query and matching advertisement bid themselves (e.g., ads previously shown to this particular user, and additional *exclusion phrases* that may be specified with each ad and are used to exclude ads if they match (part of) the query). This means that many optimizations common in information retrieval that rely on the total "score" (and in turn the rank) of an document/advertisement to be a monotonic function the scores of the matching keywords (e.g., [1], [2], [3], [4]) cannot be applied.

Finally, the fact that the indexed ad phrases are very short itself implies that even very large corpora of advertisements can be indexed in main memory. Consequently, we will describe solutions that assume all advertisement-data to be memory-resident subsequently.

C. Processing Broad-Match Queries using Inverted Indexes

To give an idea of how our approach improves the processing of broad-match queries, first consider the processing broad-match queries using the unmodified inverted indexes discussed in Section I; here, we first retrieve the union of the ad postings in the indexes corresponding to the keywords in the query (we refer to these as *candidate advertisements*), and then test if the phrase associated with each of these candidates does not contain any non-query keywords. The main problem with this approach is that the single-word indexes don't filter the advertisements well: most of the candidate advertisements will not be part of the final broad-match result, since they contain additional keywords.

However, because of the broad-match semantics, there is no need to index *all* keywords in an advertisement phrase. If we only index the keyword in each advertisement-phrase that is most rare (i.e. appears the fewest times in the ad corpus), the strategy above continues to produce the correct result and performs much better, as significantly fewer phrases have to be examined explicitly. Because broad-match queries will retrieve an advertisement that only contains a subset of the words in the query, we do not need to index advertisements *redundantly* (i.e. having the posting corresponding to a specific advertisement appear in more than one index). Using this non-redundant indexing improves the processing of any query that contains only rare keywords. However, for queries containing words common in the advertisement corpus (e.g., "*cheap movies*"), this is still very inefficient, since again many candidate advertisements will be produced.

Another modification that allows us to process broad match using the inverted indexes only (without looking up any phrases explicitly) is to – together with the ID of an advertisement – encode the number of keywords in the corresponding bid in the inverted index itself. We can then compute the bids matching a broad-match query by merging all inverted indexes corresponding to keywords in the query while keeping track of the number of times each bid has occurred. If this number is identical to the encoded number of keywords for a bid, we know that all keywords in the bid occurred in the query as well, and the corresponding advertisement matches the query.

Unlike the un-modified inverted index, this approach requires no direct access to the advertisement bids themselves; however, the second issue discussed above remains: for queries with keywords occurring frequently in the advertisement corpus, large numbers of postings have to be traversed and intersected/merged, even if nearly all advertisements they correspond to are not a match, as they contain additional keywords not contained in the query. We will show this

experimentally in Section VII-A. Retrieval speed, however, is crucial for advertisement matching, as web search itself has very tight latency constraints (see e.g., [5]), which are acerbated by the fact that a significant fraction of this latency is devoted to cover any network latency.

D. Our Approach

To overcome the issues discussed above, we need to reduce the number of candidates processed for queries containing frequent keywords. One approach for this is to index advertisement phrases not only by single words, but also by certain multi-word combinations. Multi-word combinations have advantages and disadvantages. As shown in Figure 2, there are very few advertisements for most multi-word combinations, which is convenient. Unfortunately, if we index longer combinations, then – when processing queries – there are also more combinations to look up (if we index advertisements non-redundantly, we have to lookup *all* indexed (sub-)sets of words in a query, which can be prohibitive for long queries). One insight we leverage here is that – due to the broad-match semantics – we can move advertisements to index nodes corresponding to subsets of the keywords in their bid phrases, while still ensuring correct match results. This gives us the freedom to select a specific subset of (all existing) word-combinations for indexing. One might expect that indexing on multiple keywords would result in a blow-up of the overall index size; interestingly, this is not the case due to the fact that we can index advertisements *non-redundantly*, i.e. each bid phrase appears in only a single location in the index.

In the following, we will describe an indexing framework that balances the number of indexed word-combinations to look up with the length of the results (candidate advertisements), using a cost-model for main-memory access to trade off between these two factors. Note that – while we focus on broad-match semantics – we require that the proposed structure is capable of indexing for phrase-match and exact-match retrieval as well.

E. Contributions

In the remainder of this paper, we will make the following contributions:
(I) We will describe a simple hash-based scheme for indexing and processing of broad-match queries. We will focus on broad-match performance in this paper, but the structure is capable of also handling other forms of matching used in ad retrieval such as *exact-match* and *phrase-match*.
(II) The main challenge with broad-match indexing is to trade off the number of keyword-sets we index (indexing more sets increases the number of lookups required against the index) and the number of candidate advertisements retrieved for which we have to explicitly check phrases. We will describe a hash-remapping scheme that leverages properties of the distribution of phrases in advertisement corpora and a cost model for main-memory access to trade off these factors explicitly and reduce the the overhead of long queries.
(III) We will show how to adapt the structure to (statistical information on) a query workload, resulting in significantly higher throughput overall.
(IV) We will briefly describe how to compress the resulting structure, trading off compression against access time by using a modification of the model introduced for finding the optimal re-mapping.
(V) We will provide an experimental evaluation using real-life advertisement and query datasets.

The remainder of this paper is organized as follows. Section II surveys related work. Section III defines broad-match semantics formally and outlines the basic architecture for broad-match processing. Section IV describes considerations in memory-resident retrieval processing. Section V describes optimizing the in-memory structure for a particular query workload. Extensions of our work are outlined in Section VI, followed by an evaluation on real advertisement data in Section VII, and conclusions in Section VIII.

II. Related Work

The information retrieval literature has studied inverted files in considerable depth, but there is relatively little work on broad match, as *subset*-matching is typically not very meaningful for text documents. Similar comments hold for the database literature, as well, where there is a considerable body of work on set-similarity (e.g., [6], [7]) and set-containment joins (e.g., [8], [9]), that join independent relations on the basis of the overlap in their set-valued attributes. However, these techniques are generally targeted at much larger sets of words/values and – in case of similarity joins – at constraints on the minimal *overlap* between sets, as opposed to *subset*-constraints found in broad-matches, making them impractical for our application.

The implementation of a retrieval framework for *contextual advertisements* in described in [3]; there the retrieval of advertisements is divided into both a semantic as well as a syntactic matching component. This solution uses a variant of the WAND algorithm [10], which is a *document-at-a-time* algorithm [11] that uses a branch-and-bound approach to derive efficient moves for cursors associated with posting lists. However, this approach relies on the final score of a match being a function that is monotonic with respect to the matching terms (and their scores) which – as we explained in Section I-B – is not the case for the classes of ranking functions we are concerned with.

The work that is most closely related to our approach has been proposed in the context of publish/subscribe systems. In a publish/subscribe framework, each query can be cast as an event compromised of the keywords it contains, where each advertisement is a subscription that is triggered by the set of keywords present in its bid phrase. The most closely related paper in this domain is [12]. This work is similar to our approach in that it models the underlying problem as a task of laying out the optimal in-memory indexing/processing structure, formulating this task as computing a structure that minimizes the expected access cost using a cost model of the in-memory latency.

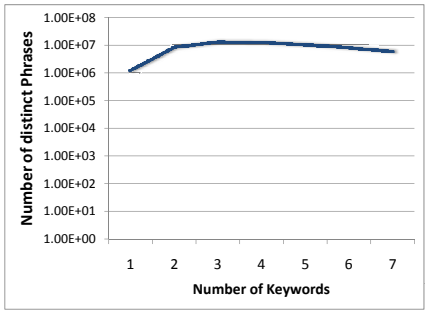

Fig. 3. Machine Translation (MT) phrases have a very different distribution than bids. Although both distributions peak at 3 words, MT phrases from the NIST competition fall off slower. MT has relatively more long phrases.

Our approach is different in a number of important ways, however. For one, we have knowledge of statistical properties of the underlying distribution of bids, which our approach leverages. Moreover, our approach does not only take sharing of predicates (= words) in ad bids into account when computing which index layout to use, but is able to leverage workload information derived from a stream of observed queries. We precisely characterize the hardness of the underlying optimization problem, and – for faster approximations – are able to give worst-case bounds on the quality of the resulting solution. Finally, the data structure we eventually use is very different from the ones best suited for pub/sub systems.

One different area where similar indexing problems have been studied is statistical machine translation (MT). MT systems construct large *rule sets*, often derived from even larger *parallel corpora*, which are bodies of text that are available in both languages. A sentence is translated by looking up each sub-phrase of an input sentence in the index, scoring the resulting rules and using a combination of the top-scoring results as the translation (e.g., [13], [14]). However, translation is different from sponsored search. Bids tend to be shorter than translation rules (using phrases of length up to 7 or more is common in settings where translation quality is essential). Figure 3 plots the distribution of translation rule lengths (up to 7) from the parallel corpus used in this year's NIST competition [15]. When comparing this to the left side of Figure 1, we can see that both distributions peak at length 3, but the drop-off is much more gradual for the NIST data.

In addition to the differences in length, MT raises some additional challenges for indexing. In MT, each translation sentence-pair may contribute to multiple rules of different lengths, meaning we cannot leverage a non-redundant indexing scheme; because of this, the indexing of such rules is often based on index structures used for longer bodies of text, such as suffix-trees or suffix arrays [16]; both of these structures are less effective in the context of broad-match queries, as a suffix-tree would increase the size of the index structure very significantly over the current solution, whereas a suffix array would result in all constant-time lookup operations into the index becoming logarithmic in the size of the corpus. However, our main ideas are applicable to tree-like structures as well, something we discuss in Section III-B.

III. Processing Broad-Match Queries

A. Notation

In the following, we denote the corpus of all *advertisements* as $\mathcal{A} = \{A_1, \ldots, A_n\}$, where each A_i corresponds to an advertisement. For each A_i, we use $phrase(A_i)$ to denote the phrase/bid associated with the advertisement A_i, and $info(A_i)$ to denote the associated meta-data (e.g., listing ID, campaign ID, bid price, information on competitive exclusion, etc.). We will use the terms bid and ad phrase interchangeably throughout the paper.

We use the notation $size(A_i)$ to denote the in-memory storage required for A_i in bytes; similarly, we use $size(phrase(A_i))$ and $size(info(A_i))$ to denote the size requirements of A_i's subcomponents. For each $phrase(A_i) = w_1 \circ \text{' '} \circ w_2 \ldots \circ \text{' '} \circ w_k$ (with $\circ$ denoting the concatination operator), we define $words(A_i) = \{w_1, \ldots, w_k\}$ as the set containing all words in the corresponding phrase; $words(p)$ for a phrase p is defined in the same way. We use $\mathcal{W}$ to denote the set of all words in the corpus and $2^{\mathcal{W}}$ as its power-set. Because – for purposes of broad-match processing – word order is irrelevant, we model a search query $\mathcal{Q}$ as a *set* of words from $\mathcal{W}$.

Broad-Match Semantics: The semantics of broad-match queries are defined as follows: given a query $\mathcal{Q} = \{w_1, \ldots w_q\} \subseteq \mathcal{W}$, retrieve all advertisements $A_i \in \mathcal{A}$ for which $words(A_i) \subseteq \mathcal{Q}$.

B. A Framework for Broad-Match Processing

A simple approach to process broad-match queries is now to index the set of words in each ad phrase using a hash table (or a similar associative data structure) and to process queries by retrieving the entries associated with all subsets of words in a query. Because the distribution of search queries is skewed towards short queries, a large fraction of these queries will result in only a small number of lookups against the hash-table; however, the number of word-subsets in a query grows exponentially with the number of words it contains, leading to poor performance for longer queries. The following discussion starts with this simple approach, we will address the issue of long queries in Section IV-B.

To hash sets of words, we define a hash function $wordhash : 2^{\mathcal{W}} \mapsto \mathbb{N}$. We use a hash table H to index each advertisement A_i, using $wordhash(words(A_i))$ as the hash key; as the hashed value we use a pointer (or an encoding of an offset) to a variable-length node storing all phrases mapped to this hash key as well as their meta-data (see Figure 4); we refer to these as *data nodes*. We use the notation $H[words(A_i)]$ (dropping the $wordhash$ function, which is implicit) as a shortcut to denote the data node that contains the metadata of A_i; we refer to the set $words(A_i)$ as the *node locator* for this data node.

Within a data node, we keep all phrases ordered by the number of words they contain (this optimization will become important later). Note that it is necessary to represent the phrases themselves due to the possibility of hash collisions between different sets of words in the ad corpus; also, they are required to process *phrase-match* and *exact-match* queries. In case that advertisement metadata is shared between many ads and/or requires significant amount of space, it is also possible – instead of storing the metadata in the data nodes – to store a pointer to the shared metadata there.

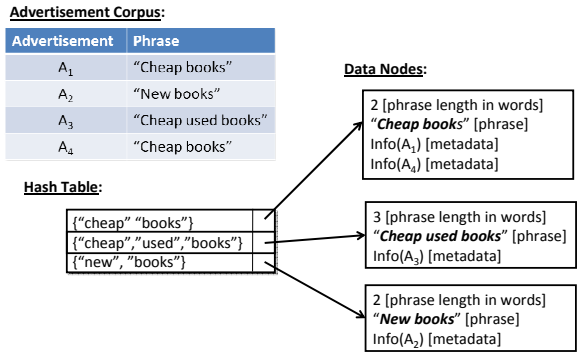

Fig. 4. The in-memory realization of the example ad corpus: sets of words are indexed via a hash-table and all advertisements sharing the same word-set are organized in a *data node*.

Now, to process a broad-match query $\mathcal{Q}$, we generate the hash-value $wordhash(q')$ for all subsets $q' \subseteq \mathcal{Q}$ and – using the hash table – visit the corresponding nodes and return all listings in them associated with ads A for which $words(A) \subseteq \mathcal{Q}$. For very short queries, this scheme will perform well, as there are only few hash-lookups required. However, for longer queries the number of lookups against the hash table grows exponentially with the number of words in the query. We will address this issue in Section IV-B.

Processing other query types: Because the structure retains information about the precise phrase associated with an ad listing, it can trivially also be used to process other match-types used in sponsored search, such as *exact-match queries* or *phrase-match* queries in which the order of words in the phrase/query has to be observed as well; the lookups against the hash table can proceed as before, only the logic to match the query against the phrase stored in the data node has to be modified.

A special case for broad-match is posed by phrases/queries that contain multiple occurrences of the same query term. As search query users tend to issue short, non-redundant queries, multiple occurrences of the same word typically carry meaning. For example, the string "*Talk Talk*" likely refers to a pop band of the same name and should not be matched to a bid containing only "*Talk*"; hence, the correct semantics for multiple word occurrences in broad-matches is defined to be that any word occurring multiple times should occur with the same frequency in both the query and the ad phrase. Hence, we treat multiple occurrences of a word as a special single word (e.g., two occurrences of the word "*Talk*" become a single word "*Talk_Talk*") in both bids as well as queries for the purpose for broad-match processing. This way, the semantics we have described above carry through directly.

Tree-structured lookup tables: Note that it is possible to use the same re-mapping scheme in cases where the associative data structure used is a tree as opposed to a hash-table, provided it supports variable sized data at the nodes (or via pointer indirection) themselves.

IV. Main-Memory Retrieval

In order to optimize data structures and processing algorithms, we need to model memory access cost. The cost model distinguishes between sequential and random memory accesses, since this difference is important in practice. Note that the broad-match retrieval problem is also almost entirely constrained by main memory latency in practice; the computation that is performed to match phrases is straight-forward.

A. The Cost of Memory Access

Explicit modeling of the latency of random data access has typically been done in the context of disk resident data (e.g. in database systems), where the disk seek overhead and the rotational delay result in a several orders-of-magnitude difference in cost between random and sequential data access. However, similar tradeoffs exist for data stored in main memory, especially if – as in our scenario – the vast majority of data is not cache-resident. Random access into memory may incur latency because of a number of factors such as: (a) L1 and L2 cache misses, (b) misses in the *translation lookaside buffer* (TLB) which maps virtual memory addresses to physical ones and (c) – in DRAM architectures – the fact that random accesses cannot use the burst-read mode of DRAMs. We also found that the new structures proposed affect the number of branch mispredictions by the processor. We will describe an evaluation of these factors in our experiments in Section VII.

In order to assess the expected access latency of a data-structure without actually executing a query workload on it, we use a simplified cost model to approximate these factors. Here, we do not explicitly model the effect of cache misses, as these depend on the memory locations accessed previously. We do model the remaining factors by assigning the "cost" $Cost_{Random}$ to a random access and a "cost" $Cost_{Scan}(m)$ to a sequential access of m bytes, once the random access to the start of the sequence has been performed.

The precise nature of the cost function depends on the processor architecture and memory chips used (see [17] for examples) and – given that e.g., the latency induced by TLB misses can vary significantly – will only be an approximation of the real behavior. Still, as we will demonstrate in Section VII, using this cost model will allow us to optimize the data structures used in broad-match processing sufficiently to obtain significant improvement in throughput when measured experimentally. Our approach is independent of the precise formulation of $Cost_{Scan}(m)$; we only require that it returns positive values and is monotonically increasing in m (i.e. $m' < m'' \Rightarrow Cost_{Scan}(m') \leq Cost_{Scan}(m'')$).

B. Reducing Latency

Because the actual computation required for broad-match processing itself is negligible, the main bottleneck of this method is the latency of main memory access, especially random access. We can reduce this latency by either (a) traversing fewer data nodes or by (b) fewer hash-lookups against the hash-table itself.

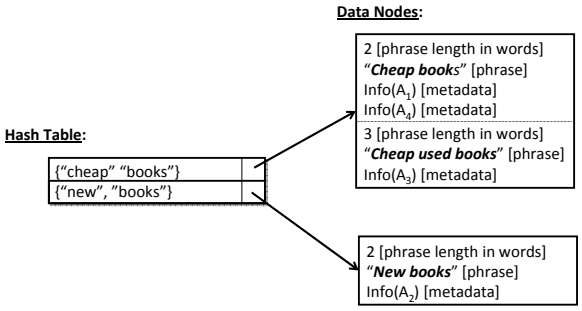

Fig. 5. The data structure from Figure 4 after A_3 has been re-mapped to the data node containing A_1 and A_4.

Reducing the number of data nodes traversed: Broad match considers all subsets of a query Q. Thus, there is an opportunity for sharing if we have two ads, A and A', where set of words in the phrase associated with A' is a subset of the words in the phrase associated with A: $words(A') \subset words(A)$. Now, if we remove the data of ad A from the node at $H[words(A)]$ and store this data at $H[words(A')]$ instead, then the result of the broad-match query will remain unchanged. Figures 4 and 5 illustrate this: moving *cheap used books* under *cheap books* saves space by eliminating an entry in the hash table. The re-mapping also takes advantage of sequential memory accesses by moving the values for the superset, *cheap used books*, to appear immediately after the subset, *cheap books*. Any query that accesses the superset, *cheap used books*, will by default also have to access all subsets including *cheap books*, due to broad-match semantics.

We refer to this as *re-mapping* data nodes. Now, if we remove all advertisements from the data node associated with $words(A)$ and distribute them to one (or more) different existing nodes in this manner, we will (i) save a random access when processing any query Q with $words(A') \subseteq Q$ and (ii) reduce the number of entries in the hash-table by one, in turn saving space and potentially increasing access locality.

Re-mapping reduces the number of random accesses at the cost of additional sequential data reads at the nodes we have moved data to. Hence we require the cost model described in Section IV-A and some notion of the relative frequency at which different phrases are queried to quantify this tradeoff.

Reducing the number of hash lookups: In the worst case, the number of hash lookups grows exponentially with query length, which would be prohibitive for long queries. However, if we re-map *all* long phrases to data nodes with node-locators of length no more than k, then in the worst case, for a query of length q, the number of hash lookups is bounded by $\sum_{i=1}^{k} \binom{q}{i}$, as opposed to $2^q - 1$. This is a big improvement in practice since $\sum_{i=1}^{k} \binom{q}{i} \ll 2^q - 1$, though realistically, we will still need to a heuristic cutoff for extremely long queries because even $\sum_{i=1}^{k} \binom{q}{i}$ is prohibitive, when q is very large. We know that the number of long bid phrases is small, which means that re-mapping all such ads to data nodes with shorter node locators will not force us to have any very large data nodes, if we chose the re-mapping destinations well. In the following, we will refer to the parameter denoting the maximum number of keywords that a phrase may have without being re-mapped to a shorter node-locator as max_words.

V. Optimizing the Index Structure

As described in Section IV-B, the reduction in the number of data nodes (and number of random accesses) comes at the cost of increasing the average amount of data we have to access per data node visited. To find the optimal tradeoff between these factors, we need to leverage both the cost model introduced above as well as information on which data nodes are frequently co-accessed by the same query (merging data nodes that are always co-accessed will not introduce any redundant memory access, merging data nodes that are rarely co-accessed likely will). To obtain co-access information, we leverage (a sample of) the query workload. In the following, we will give a definition of the resulting optimization problem, discuss its complexity and approximation algorithms.

Characterization of the Query Workload: To obtain a characterization of the overall workload, we can observe a stream of queries for an interval of time and treat it as a sample from the overall (unseen) workload. Because search query frequencies are known to follow a power-law distribution, the top most frequent queries can be identified robustly from even a small sample. These queries in turn are of most importance to the optimization problem, as any re-mapping decision involving the respective data nodes will have a much bigger impact on the overall throughput compared to other data nodes. Formally, we define a *query workload* as a set $WL = \{Q_1, \ldots, Q_h\}$ of queries. We use the function $frq(Q_i)$ to denote the respective frequency to each query Q_i in WL.

A. Problem Statement

Formally, the task is now to define a mapping of all advertisements in $\mathcal{A}$ such that the expected cost of executing the workload WL is minimized. Here, we define a mapping $M : \mathcal{A} \mapsto 2^{\mathcal{W}}$ as an assignment of advertisements A_i to the node locator they have been mapped to, i.e. the data for advertisement A_i is located at the data node $H[M(A_i)]$. For the purpose of modeling the problem, we will ignore the issue of hash collisions induced through the $wordhash$-function, as these are very rare and complicate the formulation unnecessarily. In order to preserve the correctness of broad-match queries, we only consider mappings M for which all advertisements A_i are mapped to sets to words that are subsets of the words occurring in the phrase associated with A_i, i.e. $M(A_i) \subseteq words(A_i)$. Furthermore, we require that all advertisements that share the same set of $words(A_i)$ have to be mapped to the same data node, as this allows for effective

early termination of the lookup algorithm. As noted above, the mapping also has to obey the constraint that all "long" phrases are mapped to "short" node locators, i.e. $\forall A_i \in \mathcal{A} : |M(A_i)| \leq max\_words$. For ease of exposition we assume in the following discussion that for each "long" phrase in the ad corpus at least one ad with a corresponding "short" phrase exists in the corpus (if this is not the case, such additional node-locators can be inserted easily).

We model the cost of executing a query Q under a mapping M as follows: we assume that looking up the address of a data node via the hash-table H requires a single random access and reads mem_hash bytes; accessing a data node indexed via the hash table itself requires another random access. As described in Section III-A, all advertisements are ordered by the number of words in their phrases in a data node; this means, whenever we encounter a phrase containing more words than Q in a data node, the remainder of this node is irrelevant for this query. The cost of executing Q is now simply the cost of the resulting random and sequential main memory accesses.

Under these assumptions, we define the cost of executing a workload WL given the data structure resulting from a mapping M of advertisements as the sum of the costs of all queries. We split this total cost into two components: first, we define the cost of accessing the hash table H that retains all pointers to the data nodes as

$$Cost_{Hash}(WL, M) = \sum_{Q \in WL} frq(Q) \cdot \Big(\overbrace{\min\{2^{|Q|} - 1, \tbinom{|Q|}{max\_words}\}}^{\text{\# lookups on } H \text{ required by } Q} \cdot Cost_{Random} + Cost_{Scan}(mem\_hash) \Big).$$

Second, we model the cost of accessing the data nodes themselves as

$$Cost_{Node}(WL, M) = \sum_{Q \in WL} frq(Q) \cdot \Bigg(\overbrace{\sum_{\substack{A_i \in \mathcal{A}: M(A_i) \subseteq Q \\ |words(A_i)| \leq |Q|}} Cost_{Scan}(size(info(A_i)))}^{\text{Cost of reading metadata}} + \overbrace{\sum_{\substack{p \in \bigcup_{A_i \in \mathcal{A}: M(A_i) \subseteq Q} phrase(A_i) \\ |words(p)| \leq |Q|}} Cost_{Scan}(size(p))}^{\text{Cost of reading phrases}} + \overbrace{Cost_{Random}}^{\text{Cost of random access to data node}} \Bigg).$$

The total cost then is the sum of the components:

$$Cost(WL, M) = Cost_{Hash}(WL, M) + Cost_{Node}(WL, M).$$

We define $Cost_{Node}(WL, M, N)$ as the cost of accessing data at the node with node locator $N \subseteq \mathcal{W}$ only; if no ad is mapped to the corresponding node, then $Cost_{Node}(WL, M, N) = 0$. The problem of minimizing latency can now be formulated as: *Given a workload WL and costs $Cost_{Random}$ and $Cost_{Scan}(m)$, find a mapping M such that $Cost(WL, M)$ is minimal:*

$$\operatorname*{argmin}_{M} Cost(WL, M). \tag{1}$$

We will now describe how this problem can be cast as an instance of the *weighted set cover* problem. Subsequently we will describe the algorithms and data structures we leverage to solve the resulting optimization task. In the following, we will assume that the workload is structured in such a way that each advertisement in the corpus is accessed at least once.

First, note that we can express a mapping M of $\mathcal{A}$ to h distinct data nodes via a set $\mathcal{S}_M = \{S_1, \dots S_h\}$, where each S_i corresponds to a subset of $\mathcal{A}$ containing all ads that are mapped to the same data node. In order to result in a valid mapping, $\mathcal{S}_M$ has to satisfy the requirements for mappings we have set up before:

(I) All ads must be mapped to (at least) one data node: $\forall A' \in \mathcal{A} : \exists S \in \mathcal{S}_M : A' \in S$.
(II) No ad may be mapped to more than one node: $\forall S', S'' \in \mathcal{S}_M : S' \neq S'' \Rightarrow S' \cap S'' = \emptyset$
(III) Ads may only be mapped to an existing node, whose node locator is a subset of the words the ads contain: $\forall S' \in \mathcal{S}_M : \exists A' \in S' : \forall A'' \in S' : words(A') \subseteq words(A'')$
(IV) All ads containing identical sets of words are mapped to the same data node:$\forall A', A'' \in \mathcal{A}$: if $words(A') = words(A'')$, then $A' \in S \Rightarrow A'' \in S$.

If all these conditions are met, it is straight-forward to construct the corresponding mapping M from $\mathcal{S}_M$. Using this notation, we now describe the problem of computing the set $\mathcal{S}_M$ that corresponds to the mapping minimizing the value of $Cost(WL, M)$. For this purpose, we ignore the component $Cost_{Hash}(WL, M)$ in $Cost(WL, M)$, as its value is independent of M, and only consider the contribution of $Cost_{Node}(WL, M)$.

Now, we can cast the problem of finding the optimal mapping M as the task of finding the corresponding set-representation $\mathcal{S}_M$ as a subset of a *base set* $\mathcal{S}_{Base}$ containing all possible combinations of ads that can be mapped to one data-note. We first define $\mathcal{S}_{Base}$ as the set of all subsets of $\mathcal{W}$ for which the conditions (III) and condition (IV) defined above hold. For each $S \in \mathcal{S}_{base}$ we define $weight(S)$ as the corresponding value of $Cost_{Node}(WL, M, N_S)$, i.e. the contribution of the node corresponding to S to $Cost_{Node}(WL, M)$:

$$weight(S) = \sum_{Q \in WL} frq(Q) \cdot \Big(\sum_{A_i \in S : |words(A_i)| \leq |Q|} Cost_{Scan}(size(info(A_i))) + \sum_{\substack{p \in \bigcup_{A_i \in S} phrase(A_i) \\ |words(p)| \leq |Q|}} Cost_{Scan}(size(p)) + Cost_{Random} \Big). \tag{2}$$

Now, computing the 'optimal' mapping corresponds to picking the subset $\mathcal{S}_{min} \subseteq \mathcal{S}_{Base}$ such that $\bigcup_{S \in \mathcal{S}_{min}} S = \mathcal{A}$ and $\sum_{S \in \mathcal{S}_{min}} weight(S)$ is minimal among all subsets of $\mathcal{S}_{Base}$ that cover $\mathcal{A}$, which is an instance of the the *weighted set covering problem*. To show this equivalence, we will have to show that

(i) any such set $\mathcal{S}_{min}$ is indeed a valid mapping and (ii) there is no valid mapping that results in lower $Cost_{Node}(WL, M)$.

To show that $\mathcal{S}_{min}$ is a valid mapping requires that it obeys the 4 conditions outlined above. Conditions (III) and (IV) follow directly from the fact that they hold for all sets in $\mathcal{S}_{Base}$. Condition (I) follows from the fact that $\bigcup_{S \in \mathcal{S}_{min}} = \mathcal{A}$. Finally, to show condition (II), first note that $S' \subseteq S''$ implies that $weight(S') < weight(S'')$, as equation (2) contains two summations over all (distinct) members of a set S and we assume that each ad is accessed at least once. Now, if there were a violation of condition (II), i.e. there is an advertisement $A_{duplicate}$ s.t. there are two sets $S_1 \in \mathcal{S}_{min}$ and $S_2 \in \mathcal{S}_{min}$, both of which contain $A_{duplicate}$, we will show that this implies that $\mathcal{S}_{min}$ is not a *minimal* set covering, leading to a contradiction. Here, we need to differentiate between two cases:

Case 1: $A_{duplicate}$ does not correspond to the node locator in either S_1 or S_2 (or both). Then we can create sets S_1' and S_2' by removing $A_{duplicate}$ (and all advertisements A' for which $words(A_{duplicate}) = words(A')$) from one of the sets where $A_{duplicate}$ does not correspond to the node locator. Now, we create $\mathcal{S}'_{min}$ where we replace S_1, S_2 by S_1', S_2' (unless the set we created by removing ads is empty, in which case we ignore it). S_1', S_2' (unless empty) must be in $\mathcal{S}_{base}$ and together cover the same sets advertisements as S_1 and S_2 (due to the construction of $\mathcal{S}_{base}$, all removed ads must be in both sets). Hence $\mathcal{S}'_{min}$ is also a *covering*. However, because of the subset-condition outlined above, it has lower weight, leading to a contradiction.

Case 2: If $A_{duplicate}$ corresponds to the node-locator in both nodes, then any query $q \in WL$ which accesses one of the corresponding nodes must also access the other. Hence, we can construct a $\mathcal{S}'_{min}$ by removing S_1 and S_2 from $\mathcal{S}_{min}$ and adding a set $S_{12} = S_1 \cup S_2$. This mapping must consequently cover all ads and also have a lower weight than S_1 and S_2 combined (as we "save" a random access for any query accessing S_{12}). Again, this implies that $\mathcal{S}_{min}$ is not minimal, leading to a contradiction. Consequently, condition (II) must hold.

To show that there is no valid mapping M' that results in lower cost than the mapping M constructed from $\mathcal{S}_{min}$, consider such a mapping M'. Now, we can construct the corresponding set-representation $\mathcal{S}'_M$ by starting with an empty set and adding all subsets of $\mathcal{W}$ that correspond to the sets of advertisements mapped to a single node by M':

$$\mathcal{S}'_M = \bigcup_{N \subset \mathcal{W}} \{ \bigcup_{A_i \in \mathcal{A}} M'(A_i) = N \}.$$

We assign $weight(S)$ to every subset of $S \in \mathcal{S}'_M$ as before. Now, it holds that every $S \in \mathcal{S}'_M$ is also in $\mathcal{S}_{Base}$, as – since M' is a valid mapping – conditions (III) and (IV) must be satisfied for all members of $\mathcal{S}'_M$. For the same reason, $\mathcal{S}'_M$ covers $\mathcal{A}$. However, then $\sum_{S \in \mathcal{S}'_M} weight(S) < \sum_{S \in \mathcal{S}_{min}} weight(S)$, which is a contradiction to the fact that $\mathcal{S}_{min}$ is the minimum-weight set cover. Hence no such set $\mathcal{S}'_M$ and thus no such mapping M' can exist.

B. Approximating the Optimal Mapping

Unfortunately, the fact that computing the optimal mapping corresponds to weighted set cover does not help us with regards to proposing a fast algorithm to determine this mapping. Solving the general set cover problem is known to be NP-hard, and no polynomial-time algorithm can achieve an approximation factor of better than $O(\ln |\mathcal{S}_{base}|)$ for general set-cover [18].

However, it is possible to leverage an observation on the subsets in $\mathcal{S}_{Base}$ that may become part of the final mapping to come up with a fast approximation algorithm with tighter bounds. The key insight here is that the size (in terms of number of advertisements) of any member of $\mathcal{S}_{Base}$ that can be part of the optimal mapping is constrained: any time a node contains a set of bids S such that there exists an advertisement $A_i \in S$, $words(A_i) \leq k$ for which the cost of scanning the data associated with A_i for advertisements that are stored "later" in the node which are not supersets of A_i is more than the cost of a random access (i.e. $weight(S) > weight(S - \{A_i\}) + weight(A_i)$), S cannot be part of the optimal solution. Because the difference in latency between random and sequential access in main memory is much less pronounced than it is for disk-resident data (even when aggravating factors such as TLB misses are taken into account), this – for typical weight-distributions – limits the size of a data node to a small number of advertisements.

Let k be the maximum number of advertisements that we can group in a single data node without violating the above constraint. Now, in cases where the set-size for a set-cover problem is limited by k, it is well known [19] that a simple greedy algorithm is a H_k-approximation algorithm for the weighted set cover, where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ is the k-th harmonic number, in turn giving us a simple and fast algorithm with a much tighter approximation bound than the aforementioned $O(\ln |\mathcal{S}_{base}|)$. In fact, given that we require that all advertisements for which $words(A_i)$ are identical to be mapped to the same data node, they can be considered a single element in the set (since they always co-occur in sets of $\mathcal{S}_{Base}$), meaning the approximation guarantee becomes $H_{k'}$, where k' is the maximum number of distinct $words(A_i)$-combinations in a data node. Finally, it can be shown that through the use of "withdrawal steps" this approximation-factor can be reduced further [20].

VI. Extensions

Compression: The proposed structure is very amenable to compression; we differentiate between the compression of the data nodes and the compression of the hash-lookup table.

Data-Node compression: The re-mapping techniques described earlier group together phrases in data nodes which have at least one and often multiple words in common, and are always accessed sequentially (within the node). This makes it possible to compress the phrases (in addition to any character-encoding) by representing them relative to phrases stored before them in the same data node, yielding further reduction in required storage space and access costs. In addition, sequences

of bid-prices may be compressed using delta-compression. Because any such gains are limited to a data node, they can be integrated easily with the cost model we use for the optimization problem of Section V-A by modifying the $weight(S)$ function to account for any compression.

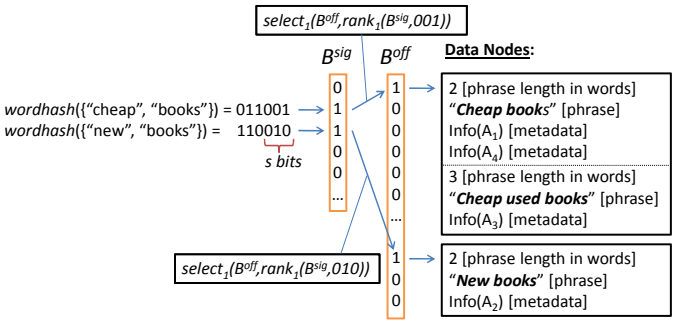

Fig. 6. Using compressed bit-arrays for lookup.

Compression of the hash-lookup table: For this purpose, it is possible to leverage *compressed binary sequences*, which have been studied in the context of compressed full-text indexes [21]; these encode compressed binary strings B while allowing a number of operations in asymptotically optimal time – the operations we require are:

- $B[i] \Leftrightarrow$ Returns the value of the i-th bit of B.
- $\text{rank}_b(B, i) \Leftrightarrow$ Returns the number of bits $b \in \{0, 1\}$ in the i-bit prefix of B.
- $\text{select}_b(B, j) \Leftrightarrow$ Returns the position i of the j-th occurrence of bit $b \in \{0, 1\}$.

The compressed representation of a bit-array B of length n containing k 1-bits requires space of $nH_0(B) + o(k) + O(\log\log n)$ bits [21] (with $H_0(B)$ denoting the *zero-order empirical entropy* of B), which is close to the optimal bound (see [22]). The structures can be used to encode a compressed representation of H as follows.

First, we use a compressed bit-array B^{sig} of length 2^s to describe all $wordhash()$ signatures for which we are storing data nodes; we set bits in B^{sig} as follows: the i-th bit is set to 1 if there exists a data node with node-locator W s.t. the s-bit suffix of $wordhash(W)$ is equal to i; otherwise, the bit is 0. We store the corresponding data nodes in main memory in order of the s-bit suffix of the hash value of their node locator; data nodes with identical suffixes are merged, similarly to the merging of data nodes discussed previously.

We encode the position of data nodes in memory using a second compressed bit-array B^{off}: let D_{size} be the size of all data nodes in memory (in byte) and D_i^{off} denote the byte-position (relative to the start of the data node storage) at which the i-th data node "starts" in memory. Now, we set the j-th bit in B^{off} to 1 if $\exists i : D_i^{\text{off}} = j$, 0 otherwise. The resulting data structure is shown in Figure 6.

We can then use these bit-arrays to achieve the functionality of H as follows: if we want to look up the data node associated with a node-locator W, we first compute sw, the s-bit suffix of $wordhash(W)$; now, if $B^{sig}[sw] = 1$ we know that there exists a data note whose node-locator has the identical suffix. To locate this data node, we now compute its offset as $offset = \text{select}_1(B^{\text{off}}, \text{rank}_1(B^{sig}, sw))$ and access the data node at this location. To see that this computation will give us the correct offset, consider that $\text{rank}_1(B^{sel}, sw)$ will return the rank of the data node with hash-suffix sw (according to the order of hash-suffixes) (i.e. the number of data nodes with a smaller suffix plus 1); now, for a given rank r, $\text{select}(B^{\text{off}}, r)$ will give the offset in memory of the start of r-th data node, which in turn corresponded to the r-th hash suffix (because of the ordering of the data nodes).

Selecting the suffix-size s: Selection of s for large bit-arrays involves an optimization task related to the one described in Section V-A: a shorter suffix causes more collisions between suffixes, in turn requiring larger data nodes to be read on average. In turn, they also result in a smaller bitmap B^{sig} and fewer 1-bits in B^{off}, reducing their size (in case of B^{off} after compression). We can use a cost-model similar to the one used earlier to model the data access latency, with the difference being that (a) we cannot control collisions/merges on the level of individual nodes and (b) the tradeoff is between the size of the structures and the access speed and not a pure optimization of access time.

In practice, these structures are able to reduce the memory footprint of H significantly for a number of reasons: (a) the bit-arrays (via a choice of a smaller suffix) can be made much smaller than the signature-information stored by hash-tables, (b) when the entropy of a bit-array is large, they can be compressed further, yielding significant gains and (c) unlike many hash-table implementations, there is no need to require the $wordhash$ signatures or the pointers/offsets into the node table to be a multiple of 8 bits in length.

To illustrate this, consider the following example. Since we assume that our techniques will be applied to corpora with very large numbers of advertisements, we will concentrate on the $nH_0(B)$-term in the space-requirement. Consider an ad corpus of 100 million advertisements, containing 20 million distinct sets of words (= $wordhash$ signatures). We assume that a $wordhash$-value requires 4 byte in memory and the lookup of the data nodes is done via a 4-byte offset into memory. Assuming that storing the data in a hash-table causes a blow-up of $\frac{4}{3}$ of the data, the in-memory size of H becomes $size(H) = (10^8/5) \cdot (4 + 4) \cdot \frac{4}{3} \approx 2.1 \cdot 10^8$ byte or $bit\_size(H) \approx 1.7 \cdot 10^9$ bits. Now, consider the size of B^{sig}. If we choose s to be 28 bit, the ratio of distinct $wordhash$ signatures to distinct positions in B^{sig} is close to 1:13, meaning a small number of additional hash collisions. Now, the number of positions in B^{sig} is $n = 2^{28}$ and the number of 1-bits in these positions only $k = 10^8/5$, meaning we can upper-bound $n \cdot H_0(B^{sig}) \leq k \cdot log_2 \frac{n}{k} + k \cdot log_2\ e \approx 8 \cdot 10^7$. To calculate the size of B^{off}, we further assume that we require 75 byte on average per distinct set of words to store phrase and meta-data, meaning that the number of positions in B^{off} is $n' = 10^8/5 \cdot 75$ and the number of 1-bits in these positions again $k' = 10^8/5$ giving us that $n \cdot H_0(B^{\text{off}}) \leq k' \cdot log_2 \frac{n'}{k'} + k' \cdot log_2\ e \approx 10^8$, meaning that

the ratio $bit\_size(H) : \left(n \cdot H_0(B^{sig}) + n \cdot H_0(B^{\text{off}})\right)$ is about 9:1.

While the data structures required to achieve these space bounds above can be very complex in practice, much simpler structures can yield significant compression as well (e.g. [23]).

Maintaining the data structure under insertions/deletions: Keeping the data structure updated to ensure the correct broad-match processing in the presence of inserts/deletions is straight forward, although deletion-operations become more expensive to process as – due to the re-mapping – we cannot identify the correct data node to delete from without processing the equivalent of a broad-match query. However, as deletions are much less frequent that queries in sponsored search, this overhead should be negligible.

It is significantly more challenging to keep the mapping itself (close to) optimal in the presence of insertions/deletions, as online versions of the set-cover problem have much weaker guarantees on the approximation bounds [24]. Therefore, instead of re-computing the optimization for each insertion/deletion, this re-computation is only performed periodically (potentially on a separate machine), while – at the time of an insertion – a mapping of the newly inserted advertisement is computed using a fast local heuristic.

VII. Experimental Evaluation

Experimental Setup: All experiments were performed on a 16GB RAM PC using a 4-CPU 2.67 GHz Xeon Processor and running Windows 2003 Server. In our experiments we use a real corpus of 180 million advertisements. To evaluate our structures, we use a web search trace with 5 million queries as the workload.

A. Processing using Inverted Indexes

In the first experiment, we are looking to validate the claim that processing broad-match queries using inverted indexes only is not sufficient to achieve performance comparable to our approach. For this purpose, we implemented the two inverted-index based processing strategies outlined in Section I-C.

(I) *Unmodified inverted indexes:* Here, we construct inverted indexes that index the least frequent word (with regards to the corpus of bids) for each advertisement phrase only (i.e. we created a "posting" in the index for this word referencing the relevant phrase and metadata). We process queries using this structure by iterating over the indexes for all words in a query, checking if $phrase(A_i)$ for any advertisement A_i we encounter contains words not in the query. This corresponds to the "non-redundant" indexing described in Section I-C.

(II) *Modified inverted indexes:* In this data structure, we store ad identifiers as postings in the inverted index for each keyword in an advertisement's bid phrase. We augment each posting by storing the total number of words in the corresponding ad phrase with the posting. Now, when processing a query, we traverse the inverted indexes corresponding to all keywords in the query, keeping track of all advertisement IDs we encounter and the number of times they are seen. Any advertisement ID that occurs as many times as the total number of keywords in the corresponding phrase is a match, since it cannot contain any keywords that do not occur in the query; any advertisement that occurs less often cannot be a match.

Note the we cannot use the well-known *skipping* optimization when processing the inverted indexes [25], since we are not merely computing intersections: e.g., if a advertisement phrase contains fewer keywords than a query, it does not have be present in all inverted indexes traversed to be a match.

The key difference between these two approaches is the amount of data in the inverted indexes accessed; while the first approach reads in much fewer postings (as no ad phrase is indexed in more then one index) on average, it has to access the phrase data to filter matches subsequently, whereas the second approach requires access to the inverted indexes only.

Results: We compared the throughput of the two techniques above to our approach using a web-trace of 5 million real user queries as the workload. Here, we used a simplified version of our approach that does not use any re-mapping of nodes and no workload-adaptation. We found that the difference in throughput was very pronounced: the throughput of the new structure was 99× the throughput of the one based on unmodified inverted indexes, and over 1300× better than the one using modified inverted indexes.

To ensure that the observed performance for the modified inverted indexes was not due to inefficiencies in our index-merging algorithm, we re-ran the same experiment, with the one difference being that we never merge any indexes, but only access each required posting once, without any further processing. This resulted in a similar discrepancy of the overall throughput, with the difference in throughput still being larger than three orders of magnitude.

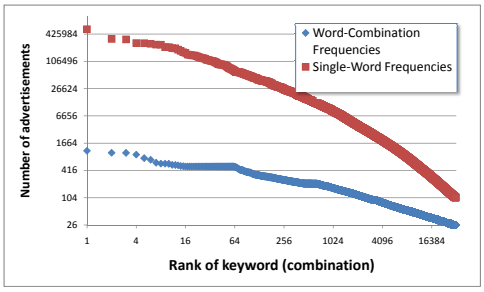

Fig. 7. Comparing the frequency-distribution of keywords to the one of word-combinations in 1.8 million advertisements.

Analysis of the performance gains: It is not hard to understand why the performance of simple inverted indexes is in order of magnitude worse compared to our approach. In the case of inverted indexes the granularity of the "buckets" we can use for filtering is limited by the number of unique terms in the corpus. Because we cannot use skip-searches for either inverted index variant, we always have to access each of these in its entirety. However, some terms are very popular in sponsored search. Using inverted indexes the number of items under keys formed by such terms is very high. It can easily exceed several thousand elements, which in turn means that – depending on which index variant we use – we

either have to check large numbers of phrases for additional keywords, or process large numbers of entries when merging inverted indexes. In contrast, with our approach the number of elements under each hash key is significantly smaller. If we focus on the most popular terms only, our experimental results show that the average number of elements under each (hash) key was reduced from about 3000 down to an average of about 100. To illustrate this graphically, we have plotted the frequency-distribution for single keywords and for the word-sets in each advertisement for the most frequent 32K keywords/combinations in Figure 7. The much larger skew of the distribution of keyword-frequencies is at the root of the observed performance problems.

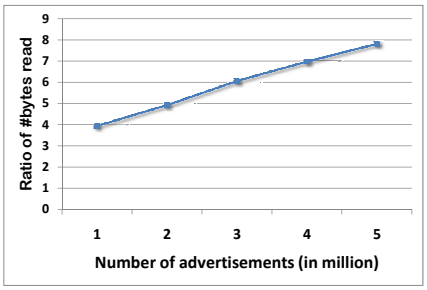

Fig. 8. The inverted-index based approach processes 4× as much data as our approach.

In order to quantify this observation, we ran an additional experiment in which we measured the total amount of data accessed by each approach for a set of 100K queries, while varying the number of advertisements in the corpus. In Figure 8 we plot the ratio between the number of bytes "read" by the processing based on unmodified inverted indexes to the number of bytes for our approach. As we can see, even for ad corpora of 1 million advertisements, the inverted-index based approach processes 4× as many bytes, and this ratio rises rapidly as we increase the size of the corpus. While the inverted-index based processing requires fewer random accesses into memory, the sheer volume of data is sufficient to account for the observed performance difference.

A similar experiment also confirmed the reason for the dramatic difference in performance for the alternative implementation using inverted indexes that encode the number of keywords in each ad phrase. Even at one million advertisements, this technique processes three orders of magnitude more data than our approach; again ratio is increasing with the size of the ad corpus.

B. Multi-Server Indexes

In scenarios where the size of the ad corpus or the index itself is too large to fit into the main memory of a single machine, it becomes necessary to split the data across servers. The bottleneck in these scenarios generally becomes the latency of the network as opposed to main memory whenever a query has to access multiple nodes to compute the final result.

To show that the performance gains of our approach are significant enough to make a noticeable difference even in this scenario, we compared our technique with the inverted-index based processing described earlier (using the unmodified inverted index, which was the faster of the two variants) in the scenario that the index and the advertisement data reside on two different servers, meaning that *every* query has to access both servers consecutively and is thus subject to network latency. As the workload we used a real query trace; we set the inter-arrival time between queries as high as possible until one of the structures did not increase in throughput.

Our experiment showed significant improvements in CPU utilization, requests per second and query response latency when compared to traditional inverted indexes, even though main memory latency is not the primary bottleneck. For example, the average CPU utilization in the case of an inverted keyword index was 98%. Using our approach the CPU utilization was reduced down to 42%. At the same time the average number of requests per second more than doubled from 2274 to 5775. Figure 9 shows the response latency distribution – for this experiment, we divided the spread of query latencies into ranges of 5 ms and computed which fraction of queries fell into each bucket (smoothing the resulting curves in the graph). As the figure shows, the average response latency improved significantly, with about 75% of the requests being processed within 10 ms compared to 32% in the case of the traditional inverted index.

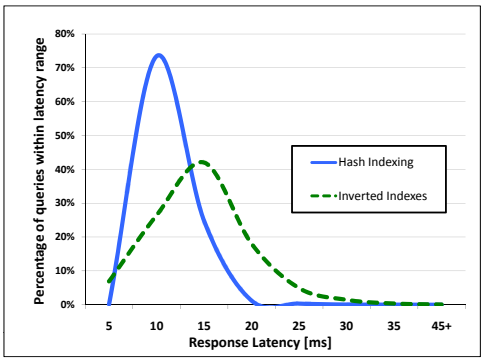

Fig. 9. Distribution of response latency.

C. Evaluation of Node Re-Mapping

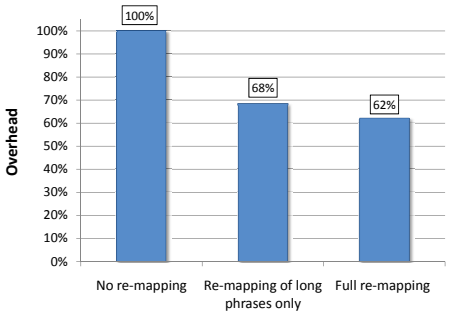

Fig. 10. Re-mapping of long queries has significant impact on throughput, optimizing the mapping yields further benefit.

To evaluate the impact of the re-mapping techniques we measured the time required to process workload of 500K distinct queries with a skewed frequency distribution (taken from the web trace via sampling) for 3 different versions of the proposed structure: (a) without any re-mapping (i.e. any query Q has to lookup and process *all* subsets of Q against the hashtable H), (b) re-mapping of "long" phrases only and (c) full re-mapping. In case of (b) and (c), we used the parameter $max\_words = 10$ and use an approximate solution to the set-cover problem to compute the mapping. Figure 10 shows the relative time taken by each structure to process the entire workload. We can see that the remapping of long phrases yields significant improvements; when the re-mapping is extended to all nodes, we gain approximately an additional 10% (relative to (b)) in performance.

Analysis of the performance gains: To understand what factors are causing the observed differences in throughput, we used Intel's *VTune* Performance Analyzer [26] tool to collect hardware performance counters for the different data structures. For this experiment, we compared the proposed data structure without any re-mapping to the data structure with full re-mapping. To compare the performance for similar access patterns, we ensure that in both cases (i.e. even when re-mapping is used) all subsets of the words in each query are looked up against the index.

Using VTune, we now measured the following hardware performance counters: (a) the total number of main memory accesses that missed the *data table lookaside buffer* (*DTLB*), (b) the fraction of unhalted core cycles spend on the page walks resulting from these misses, (c) the number of L2 cache misses and (d) the amount of cycle stalls due to branch mispredictions. Based on this data, the causes for the improved performance of the re-mapped structure appear to stem from 2 factors: for one, the amount of page-walks performed as a response to DTLB misses was significantly higher for the data structure without re-mapping (increase of more than 40%); interestingly, the number of memory accesses that missed the DTLB increased by a much smaller factor (12%). The other cause was the increased number of cache misses in the structure without re-mapping; this appears to be a consequence of the hashtable H having much fewer entries when re-mapping is performed, in turn leading to better cache locality. Interestingly, the number of branch mispredictions was larger for the structure with re-mapping (increase of 23%).

VIII. Conclusions

Sponsored search raises interesting challenges for indexing. This paper proposed an index for *broad match*, an important matching algorithm for sponsored search. Under a simple cost model of main memory latency, we reduced the optimization problem of finding the optimal index structure to weighted set cover. Although the general set cover problem is NP-hard, properties inherent in our problem statement make it amenable to fast approximate solutions. We were able to show using corpora of real ad data and query traces that our structure results in order-of-magnitude performance improvements when compared to solutions based on inverted indexes even for the simplest structure proposed, while optimizations to the main memory structure resulting from a suitable re-mapping of hash-nodes result in significant further improvements.

Acknowledgements: This paper benefitted tremendously from many insightful comments from Arvind Arasu, Raghav Kaushik, Chris Quirk, and Jingren Zhou as well as the anonymous reviewers.

References

[1] V. Anh, O. de Kretser, and A. Moffat, "Vector-Space Ranking with Effective Early Termination," in *Proceedings of the 24th annual international ACM SIGIR Conference*, 2001, pp. 35–42.

[2] T. Strohman and W. B. Croft, "Efficient Document Retrieval in Main Memory," in *30th ACM SIGIR International Conference*, 2007, pp. 175–182.

[3] A. Broder, M. Fontoura, V. Josifovski, and L. Riedel, "A Semantic Approach to Contextual Advertising," in *SIGIR*, 2007, pp. 559–566.

[4] R. Fagin, "Combining Fuzzy Information: an Overview," *SIGMOD Rec.*, vol. 31, no. 2, 2002.

[5] http://blogs.zdnet.com/BTL/?p=3925.

[6] S. Sarawagi and A. Kirpal, "Efficient Set Joins on Similarity Predicates," in *ACM SIGMOD*, 2004, pp. 743–754.

[7] A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-similarity Joins," in *VLDB*, 2006.

[8] S. Melnik and H. Garcia-Molina, "Adaptive Algorithms for Set Containment Joins," *ACM TODS*, vol. 28, no. 1, pp. 56–99, 2003.

[9] N. Mamoulis, "Efficient Processing of Joins on Set-valued Attributes," in *ACM SIGMOD*, 2003, pp. 157–168.

[10] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient Query Evaluation using a Two-Level Retrieval Process," in *CIKM*, 2003, pp. 426–434.

[11] R. Baeze-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. ACM, 1999.

[12] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems," *ACM SIGMOD Conf.*, pp. 115–126, 2001.

[13] C. Callison-Burch, C. Bannard, and J. Schroeder, "Scaling Phrase-based Statistical Machine-Translation to larger Corpora and longer Phrases," in *Proc. of ACL*, 2005, pp. 255–262.

[14] Y. Zhang and S. Vogel, "An Efficient Phrase-to-Phrase Alignment Model for arbitrarily long Phrases and large Corpora," in *Proc. of EAMT*, 2005.

[15] "NIST MT Competition," http://www.nist.gov/speech/tests/mt/.

[16] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," in *Journal of the ACM, 23*, 1976, pp. 262–272.

[17] S. Büttcher and C. L. Clarke, "Index Compression is Good, Especially for Random Access," in *In Proceedings of CIKM*, 2007.

[18] U. Feige, "A Threshold of ln n for Approximating Set Cover," *Journal of the ACM*, vol. 45, no. 4, pp. 634–652, 1998.

[19] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," *Mathematics of Operations Research*, vol. 4, pp. 233–235, 1979.

[20] R. Hassin and A. Levin, "A Better-Than-Greedy Approximation Algorithm for the Minimum Set Cover Problem," *SIAM J. Comput.*, vol. 35, no. 1, pp. 189–200, 2005.

[21] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comput. Surv.*, vol. 39, no. 1, 2007.

[22] A. Golynski, "Optimal Lower Bounds for Rank and Select Indexes," *Theor. Comput. Sci.*, vol. 387, no. 3, 2007.

[23] S. Vigna, "Broadword Implementation of Rank/Select Queries," *LNCS*, no. 5038, 2008.

[24] G. Ausiello, A. Giannakos, and V. T. Paschos, "Greedy Algorithms for On-line Set-Covering and related Problems," in *Proc. of CATS*, 2006, pp. 145–151.

[25] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes - Compressing and Indexing Documents and Images*. Morgan Kaufman Publishers, 1999.

[26] "Intel VTune Performance Analyzer," http://www.intel.com/cd/software/products/asmo-na/eng/vtune/.