Stochastic Query Covering for Fast Approximate Document Retrieval

ARIS ANAGNOSTOPOULOS, Sapienza, University of Rome, Italy
LUCA BECCHETTI, Sapienza, University of Rome, Italy
ILARIA BORDINO, Yahoo! Research, Barcelona, Spain
STEFANO LEONARDI, Sapienza, University of Rome, Italy
IDA MELE, Sapienza, University of Rome, Italy
PIOTR SANKOWSKI, University of Warsaw, Poland

We design algorithms that, given a collection of documents and a distribution over user queries, return a small subset of the document collection in such a way that we can efficiently provide high quality answers to user queries using only the selected subset. This approach has applications when space is a constraint or when the query-processing time increases significantly with the size of the collection. We study our algorithms through the lens of stochastic analysis and we prove that, even though they use only a small fraction of the entire collection, they can provide answers to most user queries, achieving a performance close to the optimal. To complement our theoretical findings, we experimentally show the versatility of our approach by considering two important cases in the context of web search: In the first case, we favor the retrieval of documents that are very relevant to the query, whereas in the second case we aim for document diversification. Both the theoretical and the experimental analysis provide strong evidence of the potential value of query covering in diverse application scenarios.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms: Algorithms, Measurement.

Additional Key Words and Phrases: Algorithms, Query Covering, Stochastic Analysis, Caching.

ACM Reference Format:
Aris Anagnostopoulos, Luca Becchetti, Ilaria Bordino, Stefano Leonardi, Ida Mele, and Piotr Sankowski. 2013. Stochastic Query Covering for Fast Approximate Document Retrieval. *ACM Trans. Inf. Syst.* 0, 0, Article 0 (0), 34 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

With the huge growth of data collections, the area of approximate query answering has received increasing attention in the database community over the past 10–20 years. The idea is that whenever a user submits a query, the system might return an answer that is not the best one according to the selected ranking criterion, but is close to it. This can be a cost-effective strategy when the approximate answer is fast to compute, bringing savings in response time or storage. It is beneficial in settings where: (1) an exact answer is not necessarily required, and an approximate response suffices to provide the necessary information, (2) a preliminary response is returned while the exact answer is calculated, or (3) fast feedback is provided to the user about the quality of the query.

This article is an expanded version of the article *Stochastic Query Covering* appeared in *Proceedings of the 4th ACM International Conference on Web Search and Data Mining*, pages 725–734, 2011.
Author's addresses: A. Anagnostopoulos, L. Becchetti, S. Leonardi, and I. Mele, Sapienza, University of Rome, Department of Computer, Control, and Management Engineering *Antonio Ruberti*, Via Ariosto 25, I-00185 Roma, Italy; email: aris@dis.uniroma1.it, becchetti@dis.uniroma1.it, leonardi@dis.uniroma1.it, mele@dis.uniroma1.it; I. Bordino, Yahoo! Research Barcelona, Avda. Diagonal 177, 08018 Barcelona, Spain; email: bordino@yahoo-inc.com; P. Sankowski, Warsaw University, Institute of Informatics, ul. Banacha 2, 02-097 Warsaw, Poland; email: sank@mimuw.edu.pl.

DOI : http://dx.doi.org/10.1145/0000000.0000000

A common technique to support approximate query answering is the creation of a suitable summary (or synopsis or sketch) of the entire dataset that is appropriate for particular applications [Acharya et al. 1999; Dobra et al. 2002; Garofalakis and Gibbons 2002]. These are created either by calculating some statistics over the dataset or by sampling some of its contents.

In this paper we address a related aspect, namely the problem of approximate document retrieval, based on query-log analysis. At a high level we have a large collection of documents and a (known) distribution over queries. For each query–document pair we introduce a score, which for instance can model the relevance of the document to the query. We are interested in returning the top-k documents for user queries that appear according to the query distribution. The goal of this paper is to design algorithms that, given the query distribution, the collection of documents, and the score of each query–document pair, return a subset of the document collection of smallest possible size, such that in expectation we are able to adequately answer new queries drawn from the distribution, where the term "adequately" will be formally defined in Section 3.1.

1.1. Techniques

To address the problem described in the previous section, we present the *stochastic query covering* problem as a suitable model of the scenario outlined above. In addition, as we discuss in Section 4.3, it allows for insightful analysis whereas a worst-case approach does not provide any intuition. The downside is that the analysis becomes technically more involved.

Specifically, we assume a framework in which users submit queries to a document retrieval system over time. The system uses a *cache* of limited size to hold a subset of the document collection. Ideally, documents in this subset tend to be highly relevant for queries that users are most likely to submit. Whenever a user submits a query, the document retrieval system must return a set of documents relevant to the query.

We emphasize that the approach we consider is *agnostic* to the adopted notion of relevance or the way it is measured by the underlying retrieval system. In particular, we assume that the underlying retrieval system, given a query, can calculate a weight for a document that measures the importance of the document to the query. Typically, the weight would be the total score that the system assigns to each document, which is subsequently used for ranking the results, such that documents with higher relevance scores appear first in the result list.

To return a result page, the system either constructs a result page using documents stored in cache, or it fails; in the latter case, the system incurs a *cache miss*, that is, a penalty reflecting the fact that constructing a result page will require a time-expensive operation, such as accessing secondary storage or retrieving contents from a back-end server.

For any given query, a document has a query-dependent weight that measures its importance with respect to a specific list of query results (e.g., weights measure the degree of relevance of the documents to the queries). Whenever a query q is submitted, the document-retrieval system must return a set of documents whose overall weight with respect to the query is at least some given threshold W_q, or as large as possible, subject to a cardinality constraint.

This weight-based approach provides a flexible way to design selection strategies that are sensitive to different optimization criteria by suitably defining weights. To showcase the versatility of our approach, in this paper we present two weight definitions that account for query–document relevance or result diversification. As to the latter, in a nutshell, *result diversification* means maximizing the probability that, upon submitting a query, a user obtains at least some relevant documents for the intent behind the query she submitted. Result diversification is of use for *ambiguous queries* (e.g., "python," "jaguar"), as well for *broad queries* with many potential user intents (e.g., "java programming language," "skype problems").

A special case that we consider is the binary setting, in which a document can be either *relevant* or *nonrelevant* to a query.[1] The document-selection process prefers to store in memory those documents that "cover" a large fraction of queries, and a query is considered covered if at least k relevant

[1]This is the version considered in the preliminary version of this paper [Anagnostopoulos et al. 2011].

documents are selected. The limitation of this approach is the binary definition of relevance, which makes the selection process completely unaware of the degree of relevance of the documents to the queries.

From an application viewpoint the approach we present here is much more powerful than the binary one, as it allows the application of our framework to realistic scenarios, where documents have different importance for the query. Indeed, in Section 5, we will see that the weighted approach provides close to optimal (with respect to quality) results, and that it is superior to the binary one. From an analysis viewpoint the problem becomes harder, as we lose part of the combinatorial structure possessed by the binary version presented in [Anagnostopoulos et al. 2011]. Furthermore, it introduces the problem of assigning the right weights to query–document pairs, an issue that is application dependent and that we discuss more in details in Sections 3.1 and 4.2.

1.2. Applications

Our approach is general enough and is applicable whenever we have a large document collection that we need to query. We assume that a query result is a set of different documents relevant to the query, and that we know (or we have enough statistics to learn) the query distribution. It is motivated by scenarios where we have small storage capabilities such as in the case of small portable devices, large communication costs, or when the query-processing time increases significantly with the size of the collection. In the remainder we present some concrete potential applications in which we believe that our ideas could be of use, either per se or in combination with other techniques.

Web search. Optimization of search-engine performance is obviously of paramount importance given that a typical search engine receives a huge number of queries every second, and users expect very low response times. To this end, search engines employ a variety of caching techniques as a means to provide results timely and with minimal reduction in quality. Caching can be *static* (results to be cached are precomputed) or *dynamic* (the cache is updated dynamically, depending on the query sequence) and usually a combination of both techniques is applied [Fagni et al. 2006]. Static caching also includes the various techniques of *index pruning* and *tiering*, where a subset of the inverted index used to retrieve documents is stored in fast memory to allow for fast retrieval.

There are two main cache architectures that a system can use, and often a combination of them is employed [Saraiva et al. 2001]. In the first architecture the cache stores results of most frequent queries, while in the second one the cache keeps posting lists (or portions of them) of most frequent query terms. The query-result cache may store the entire pages of results, so that queries can be answered very efficiently by the search system. An alternative approach is more document oriented: a cache is a collection of documents indexed appropriately, and to respond to a query the search engine can access the cache index and subsequently, if required, the main memory. A drawback of the latter approach with respect to the former is that query cache response times are higher, because result pages must be constructed from documents in the cache. On the other hand, this cache presents the advantage that the same document can be used to serve multiple queries, so that the storage space is used more efficiently. Both architectures can be employed concurrently, for example, one can use a cache storing the complete page of results for the most frequent queries, whereas a second level cache storing single results can be used if the result page for a query is not found in the first-level cache.

Our approach can be used to implement static-only or full-fledged static–dynamic caching (SDC) strategies. In the former case, it is suitable in settings where it is permissible not to return the absolute top-k documents (this might be the case in a large corporate search engine). Even though it is a type of index pruning (we can create pruned indexes from the subset of documents that we select) our motivation is different from the standard use of index pruning. Usually, when we talk about tiering and index pruning, there is the requirement that *all the top-k documents of the collection are returned for each query*, otherwise we have a cache miss [Ntoulas and Cho 2007]. Instead, in our approach, we allow (actually we often prefer), for some queries, to return highly relevant documents but not necessarily all the top-k. This allows to achieve much more savings, as we show in Section 5.

Furthermore, for queries where some documents are necessary (e.g., navigational queries), we can set the weights in such a way that we enforce these documents to be included in the results whenever a user performs these queries—more discussion is available in Sections 3.1 and 4.2. In the latter case, it can be used to implement static–dynamic caching strategies in the spirit of [Fagni et al. 2006]. In particular, in Section 5.3 we experimentally show that using our approach to implement the static-cache component of the SDC strategy presented in [Fagni et al. 2006] brings an improvement in cache performance.

Offline search. Another application could be the creation of a service that would allow for offline searching and browsing[2]. Consider, for example, a tourist visiting a new city. She might be interested in various information while in the city, such as accommodation information, transportation options, cultural or sports events, restaurants, history information, practical information, related news, and so on. A useful application would be a service that would allow her to be able to search without connection (either because of unavailability due to lack of infrastructure, or because of high connection cost) and obtain the relevant information. This is exactly the problem we consider here: for the potential queries that the user might have (gathered from historic data) the application must have stored at least a number of results; however, because the storage space is limited, the results must be selected in such a way, so as to cover the user's needs with high probability. Such an application could also use first this approach as a means to search offline and if the information retrieved is not sufficient it could attempt an online connection (if available), incurring additional cost.

Other uses. Our approach can be also employed to facilitate the access to web content. A content delivery network (CDN) [Buyya et al. 2008] is a collection of network elements arranged for more effective delivery of content to end users. They improve performance through caching or replicating content over some mirrored servers, which are strategically placed at various locations to deal with sudden spikes in the requests. A critical aspect in the operation of a CDN is how contents are placed on surrogate servers. Ideally, the contents placed on a surrogate server should be chosen so as to maximize the amount of requests served locally, without interaction with back-end CDN servers or the origin server of the requested contents. A number of techniques are used to address this issue, with the CDN provider typically pushing contents to surrogate servers based on popularity, so that each surrogate server ideally hosts contents that have appeared to be the most popular among users in the region that it serves.

Another potential application is in the area of computational advertising. Typically, when a user performs a web-search query (in the case of sponsored search) or visits a content page (in the case of content match) the online service provider (OSP), such as Google or Yahoo, selects a small number of ads to show to the user from a pool of several hundred million ads. Choosing the appropriate ads is a complicated procedure involving information retrieval systems that retrieve ads that are relevant to the page or query, auctions for choosing the ads to display among the relevant ones, and ad exchanges (RightMedia, DoubleClick Ad Exchange, etc.), which facilitate bidding, buying and selling ads from multiple ad networks. At times of high demand there is a need for caching so that the OSP can avoid the continual application of the information-retrieval step and the time consuming ad-exchange process. The exact application of the model presented here to the ad scenario requires addressing several issues that are outside the scope of the present work.

We finally believe that our approach is very general and can have a lot of applications that we are currently unaware of. In cases where the retrieval time highly depends on the collection size, a query-aware caching strategy can help to reduce it providing at the same time quality guarantees. As example of potential applications we mention semantic search (in which statistical NLP techniques might have to be applied at query time), image or video retrieval (which may involve time-consuming image/video processing), or querying of large biological databases (for gene functions, protein structures, similarities of biological sequences, etc.).

[2]Such a service was offered by the Webaroo startup (http://webaroo.com) before the company changed its focus.

1.3. Contribution

Our goal is to show analytically and experimentally that taking into account the query–document structure and statistical information from query logs can be beneficial in a variety of ways, when designing a modern document-retrieval system. Thus, the contributions of this work are the following:

— We define the query-cover problem as a means to efficiently select documents to store in a limited memory (e.g., a static cache of query results).
— We define algorithms, which we analyze in a realistic stochastic framework. Even though theoretical analysis is often neglected in the design of caching policies because of technical hardness, we believe that it reveals the possibilities and the limitations of our approach. We show that our algorithms have performance close to the optimal possible.
— We propose and analyze strategies that address two crucial and diverse aspects of document retrieval, ranking and diversification, demonstrating the versatility of the approach.
— We validate our theoretical findings experimentally, by implementing weighting schemes for ranking and diversification and comparing the resulting caching algorithms against strong baselines.

Remark. We emphasize that our goal is not to show that our approach can replace current caching techniques or to propose a new diversification strategy. Instead, we want to show both theoretically and through simulations that a system designer can use query covering for obtaining a summary of a document collection, or as a basis and as a component for the design of static caching strategies that strike a good balance between user-perceived latency and other optimization criteria. Furthermore, the approach we propose is versatile, in the sense that the pursuit of diverse optimization goals (high result relevance or result diversification being two important examples) boils down to defining suitable weights. We discuss more in Section 3.1.

Overall, we believe that the global treatment of the topic—through analysis under generic and realistic input assumptions and through extensive experiments—provide evidence that this generic approach can be tailored to particular needs in diverse application scenarios.

Roadmap. In the next section we present some related work. We define the query-cover model in Section 3, then we provide and analyze greedy algorithms in Section 4. The experimental evaluation is described in Section 5. Finally, we conclude and present suggestions for future work in Section 6.

2. RELATED WORK

In this section we describe some areas related to our work. We first present the state-of-the-art in the fields of query-log analysis, web-search-engine caching, query-result caching, tiering and pruning. Then we present works related to diversification of search results. We also describe the similarity caching and its potential use in content-match advertising, and we conclude with a brief description of theoretical works on set cover and its stochastic variants.

Query-log analysis. Several studies have analyzed query-log data to understand the search behavior of users, to improve the caching strategies, and to evaluate cache performance. Spink et al. [2001] analyzed a query log from the *Excite* search engine and observed that users typically submit short queries and look at the first pages of results. In addition, they found that users prefer to refine queries rather than to navigate answer pages deeper, and that generally they do not exploit advanced search features. Several authors analyzing query-log datasets observed a *spatial locality* in the query stream: many queries are submitted once or twice, whereas few queries are submitted many times [Silverstein et al. 1999; Xie and O'Hallaron 2002]. The frequent queries are also very popular, because they are shared by different users. Further studies emphasized that repeated submissions of queries on the same topic are separated by a small number of other queries, reflecting a *temporal locality* in the query stream [Fagni et al. 2006; Markatos 2001]. These observations justify the idea of exploiting a server-side caching system for storing results of most frequent or most recent queries: the system can precompute in the cache answers or partial data to be used at query time in the computation of new answers.

Caching in web search engines. Modern web search engines are large-scale distributed systems [Baeza-Yates et al. 2007; Barroso et al. 2003; Cambazoglu et al. 2009] in which the index is partitioned among multiple processing clusters and queries are processed over smaller indices. Serving a user query entails a number of steps [Barroso et al. 2003; Cambazoglu et al. 2009]: the front-end machine receives the query and forwards it to the back-end machines, which return the identifiers of the most relevant documents. The front-end machine merges and ranks the results by their relevance, then it uses the top-k (e.g., $k = 10$ or 100) document identifiers to retrieve urls and snippets from the document repository and build the result pages. To improve query throughput, caching can be employed at different levels [Saraiva et al. 2001]. The front-end machine caches results of the most frequent or most recent queries. At the level of the core search modules, a cache stores posting lists (or portions of them) associated with the most frequent query terms. Baeza-Yates et al. [2008] analyzed tradeoffs between caching of query results and posting lists. Finally, at the level of document repository, the cache can store copies of documents, document surrogates or supersnippets. For a survey about these techniques refer to [Ceccarelli et al. 2011].

Query-result caching. During the last decades, query-result caching has received a lot of attention. A first attempt to use past queries to enhance the retrieval process was performed by Raghavan and Sever [1995]. They exploited the user-query history to build a set of persistent "optimal" queries, which can be used to improve the retrieval effectiveness of similar future queries. Lempel and Moran [2003] studied the task of caching query result pages. They evaluated different replacement policies based on variations of *LRU* and proposed *prefetching* to predict future user requests. The proposed replacement policy, called *PDC* (*Probabilistic Driven Cache*), is based on a probabilistic model of user-browsing behavior. Fagni et al. [2006] observed that a simple popularity-based cache of query results addresses poorly the issue of temporal locality. They proposed *SDC* (*Static–Dynamic Cache*), which consists of a *static* portion for storing results of most frequent queries, and a *dynamic* portion for which an *LRU*-like strategy is applied to capture temporal changes in the query stream. Combined or not with prefetching strategies, *SDC* outperforms standard caching policies on the datasets considered by the authors. The analysis made by Fagni et al. was refined by Baeza-Yates et al. [2007] and by Gan and Suel [2009]. In the first work, the authors studied the effect of the tail of the query distribution on caches of web search engines and they proposed *admission policies* to avoid caching of not frequent queries. The admission policies use features related to queries and to usage information. In the second work, the authors considered the case in which queries have a weight, which can be used to represent the cost of serving a query. Their work aims at maximizing the overall weight of queries served entirely in cache.

Tiering and pruning techniques. An important area of research is related to the study of the multi-tier architecture and index-pruning techniques often employed by modern search engines.

Risvik et al. [2003] were among the first researchers to introduce the concept of multi-tiering. They proposed various three-tier architectures and assessed their performance. Their strategy for selecting the documents to place in the second tier as opposed to those placed in deeper (and slower) tiers is exactly the top-k heuristic considered further in this paper. In particular, we show how the strategy we propose outperforms top-k in several aspects.

Baeza-Yates et al. [2009] considered the problem of characterizing the tradeoff between average response time and infrastructure cost in two-tiering systems, when a predictor is used. Differently from their work, our aim is to design strategies to obtain the corpus of the first tier, so that, in expectation, the fraction of queries that are satisfied by the first tier is maximized.

Leung et al. [2010] assumed that a number j of tiers (of increasing sizes and costs) are available and their goal is to allocate documents to tiers, so as to minimize the sum of the costs for the queries (the case we consider corresponds to $j = 2$). We consider a different setting in which: (1) each document has different degrees of relevance for different queries, whereas a query has an associated weight corresponding to its request probability; (2) the result set for a query is part of the design, namely, the number of documents to store in the static cache is minimized and the overall relevance of documents exceeds a given threshold.

Pruning [Lam et al. 2010; Ntoulas and Cho 2007] is a related and orthogonal thread of work, in the sense that caching schemes can operate on top of pruned index schemes to reduce answer costs.

Search-result diversification. Early works on information retrieval [Boyce 1982; Goffman 1964] recognized the importance of diversifying a list of search results. Carbonell and Goldstein [1998] proposed the Maximal Marginal Relevance (MMR) method, which balances the relevance of search results with their marginal novelty with respect to the documents already included in the result list. Relevance and novelty are evaluated using two similarity functions (query–document and document–document). Other approaches inspired to MMR differ mostly by how the similarity between documents is computed [Lafferty and Zhai 2006; Rafiei et al. 2010; Wang and Zhu 2009].

Another important line of research takes into account the different semantic aspects of queries and documents. A first set of approaches does this only implicitly. Bookstein [1983] based the selection of search results on the probability of documents being relevant conditioned on the documents that had appeared before. This method requires an explicit user feedback for relevance after selecting a document. Chen and Karger [2006] formulated an objective aimed to find at least one relevant document for all users. Radlinski et al. [2008] computed an optimal ranking of search results from a diverse set of orderings (so as to maximize the probability that a user finds at least one relevant result), exploiting common clicks to penalize similar documents.

A second group of methods makes explicit use of the knowledge about the different topics covered by queries and documents. Such a knowledge can be gathered in various ways. Agrawal et al. [2009] applied a taxonomy-driven approach to categorize both queries and documents into different aspects. Santos et al. [2010] interpreted the query reformulations provided by search engines as subqueries that capture different aspects of the input query. Capannini et al. [2011] extracted subtopics from query refinements stored in search-engine logs. All these methods rerank a list of search results according to the relevance of the subtopics of the input query, also taking into account the relative importance of each semantic aspect.

Diversification was also studied in contexts different from Web search, like question–answer systems [Clarke et al. 2008], online shopping [Vee et al. 2008], and recommendation systems [Ziegler et al. 2005].

We remark that in this paper we do not intend to propose a novel technique for diversifying search results. We use a state-of-the-art approach for diversification, namely [Capannini et al. 2011], to select and store in a static cache those documents that maximize the satisfaction of the user even for the ambiguous/faceted queries. To do that, we use weights that reflect the relevance of the document to the query along with the ability of the document to satisfy a different aspect hidden behind the query keywords.

Similarity caching. Chierichetti et al. [2009] studied a related problem in contextual online advertisements associated to Web pages. In this case, upon a user's visit to a Web site, the online service provider (e.g., Yahoo or Google) should ideally choose the most relevant advertisement to display to the user based on the user characteristics and the contents of the visited page. The whole procedure can be seen as a querying process, in which the user visiting a given page is the query and serving the query means returning an advertisement that is relevant for the (user, visited page) pair. The authors proposed caching strategies for online advertisement information that are based on a notion of similarity between queries, so that a query p is satisfied in cache whenever the system returns an advertisement that was previously used for a query q that is sufficiently similar to p (according to some distance metric).

Set cover and its variants. The problem considered in this paper turns out to be a stochastic version of the well-known *set-multicover problem*, itself generalizing the *set-cover problem* [Vazirani 2001]. Both problems are NP-hard and it is known that the natural greedy algorithm provides logarithmic approximations for these problems and that this bound is tight [Vazirani 2001]. This problem has been also studied in the online setting. As in the offline case, here the input is given by a universe of items and a collection of subsets thereof. In this case, items are released over time and the online algorithm must incrementally maintain a collection of sets that covers all items released so far.

Table I. Notation Table.

Q	Set of queries (elements)
U	Set of documents (sets)
$n = \|Q\|$	Number of queries
$m = \|U\|$	Number of documents
q	A typical query
u	A typical document
Q_u	Set of queries to which u is relevant
C	Set of documents that contribute to the result list of a query
R_q	Result list of q
$Rank_{u \in \mathcal{R}_q}$	Position of u in q's result list
$\mathcal{Q}$	Probability distribution over queries
$f(q)$	Probability of query q
k	Number of documents required to cover a query (e.g., $k = 10$)
$w_q(u)$	Weight of the document u for query q
W_q (or W)	Overall weight required to cover query q

Alon et al. [2003] proved an almost tight poly-logarithmic competitive ratio for this problem, a result that was slightly improved by Buchbinder and Naor [2009]. Very often, items are released online according to some stochastic process. Grandoni et al. [2008] studied the problem under the assumption that the underlying stochastic problem is stationary, but its distribution is unknown. The authors proposed online strategies that, on average, have performance that is at most a logarithmic factor away from the offline optimum.

3. MODEL

We start by stating the setting that we consider. We consider an underlying retrieval system which, given a query, can calculate a weight for a document that quantifies the importance of the document to the query. We discuss possible weight definitions further in the paper; for now, we note that weights can reflect relevance or other notions of interest. For example, weights may just depend on the ranking (see Section 3.1), they can be assigned in a way that reinforces diversification, or they could depend on expected revenue (this might be desirable for an application such as computational advertising). For concreteness of the presentation, we often refer to the weights as a measure of the relevance of the document to the query. Furthermore, we assume that we do not have complete knowledge of the queries that are going to be submitted in the future, but we have a good estimate of their distribution.

We assume a ground set Q of n queries and a set U of m documents. For every pair (q, u), with $q \in Q$ and $u \in U$, there is associated a weight $w_q(u)$. A sequence of t queries are being drawn independently (with replacement) from the Q queries according to a probability distribution $\mathcal{Q}$. The probability of a query q is given by $f(q)$. Each query is *covered* by a set of documents, which are relevant to the query, where we define a document u *relevant* to a query q if it appears in the query's result list, hence if $w_q(u) > 0$. Conversely, each document $u \in U$ *covers* a set of queries Q_u, which are the queries to which it is relevant.

We consider a query q covered if it accumulates sufficient weight. Given a *threshold* W_q, query q is covered if and only if there is a set of documents C such that $\sum_{u \in C} w_q(u) \geq W_q$. Without loss of generality (by rescaling the weights $w_q(u)$) we can assume that there exists a unique value W such that for all queries we have that $W_q = W$.

We want to create a universal map (it can be computed at night time or, generally, at periods of low system usage) from each query to a set of relevant documents for the query, so that we can store the documents, returned from the map, in a static cache, and when the query is submitted we can fetch the documents from the cache to serve it. We now formalize our problem as follows:

PROBLEM 3.1 (QUERY MULTICOVER(t)). *Given a set Q of queries, a distribution $\mathcal{Q}$ over Q, a set U of documents, a set of weights $w_q(u) : Q \times U \mapsto \mathbb{R}_+$, a threshold W, and a sequence length t, compute a mapping ϕ from each query $q \in Q$ to a set $\phi(q) \subset U$ such that for each $u \in \phi(q)$ we have*

(1) $q \in Q_u$,
(2) $\sum_{u \in \phi(q)} w_q(u) \geq W$,

and our choice minimizes

$$\mathbf{E}\left[\left|\bigcup_{i=1}^{t} \phi(q_i)\right|\right],$$

where the expectation is taken over all the sequences of t *queries* $q_1, q_2, \ldots, q_t$ *drawn independently from* $\mathcal{Q}$.

In practice, we do not know the distribution $\mathcal{Q}$, but we use statistical information from past user transactions. Furthermore, we are able to select a set of documents which covers only a subset of the queries that are likely to be submitted. We discuss about these and other practical issues in Section 4.3.

Note also that the problem definition has the sequence length t as part of the problem input (so that the minimization problem is well defined); however, we would like to have policies that perform well even when the value of t is not known. Indeed, the algorithms that we present later create mappings that provides a small cost (compared with the optimal solution) without any prior knowledge of t, that is, the same mapping is competitive for all values of t.

3.1. Query-Document Weights

In this section, we discuss examples of weight definitions, which result in a bias of the caching strategy towards maximizing relevance of cached documents or diversification of search results.

Binary Relevance. A simple and important case is the binary one, in which each document can be either relevant or not to a query. When the user issues a query, the document retrieval system has to return at least k (e.g., $k = 10$ or 100) documents. This simple definition of binary relevance is captured by setting the query–document weights as follows:

$$w_q(u) = \begin{cases} 0 & \text{if } u \notin \mathcal{R}_q, or \\ 1 & \text{if } u \in \mathcal{R}_q, \end{cases} \tag{1}$$

where $\mathcal{R}_q$ represents the result list of query q, which in this case is the set of documents that are relevant for it. Then we set W equal to k (the overall coverage weight is exactly the number of documents relevant to the query that we want to retrieve from the cache). Although the relevance definition is very simple, it shows the potential of the smart selection of documents against the simple selection of top results of most frequent queries. The main drawback of the approach is that it is not perfectly suitable for some of applications where ranking is of paramount importance, such as web search, with a degradation of precision, as we will see in Section 5.2.

Ranking relevance. The main goal in this case is to have weights that favor the inclusion of documents that are ranked highly in a large set of queries. A natural approach is to have weights that decrease as the position of a document in the ranked list of results[3] increases (we assume that the first document appearing in the list has position 1, the second one has position 2, etc.). One reasonable way to pursue this goal is to define $w_q(u)$ as

$$w_q(u) = \frac{1}{\log_b(Rank_{u \in \mathcal{R}_q} + 1)}, \tag{2}$$

where $\mathcal{R}_q$ represents the result list of query q and $Rank_{u \in \mathcal{R}_q}$ is the position of u in the result list. Note that by varying the base b of the logarithm we can trade off the goal of including higher ranking documents with the objective of having documents that cover a large number of queries.

[3]This is the list of results that would be returned if the entire collection were considered.

This is probably the most natural weighting scheme, as weights decrease but not very rapidly. We have also used $w_q(u) = \frac{1}{Rank_{u \in \mathcal{R}_q}}$, but in our experiments the distribution of the weights is too skewed and the results are almost always dominated by a few top documents.

Other reasonable approaches might use some measure of similarity between queries and documents (e.g., cosine similarity and other related techniques in information retrieval). Yet, selecting a weight based on the rank is simple, more universal, and probably more effective, because similarity between queries and documents is already taken into account in the computation of the ranking performed by the ranking algorithm of the search engine.

Diversification. As a further example, we propose a definition of weights that biases our caching strategy towards maximizing the coverage of all possible intents behind user queries. In this case, the weight of a document not only depends on the position of the document within the result list, but also on the extent to which the document covers different intents for the query.

Queries submitted by users to a search engine typically consist of few terms. Such queries have often more than one possible interpretation and it may happen that the top results returned by the search engine do not correspond to the user's intent behind the query, so that the user refines her query (generally adding words). The sequence of queries submitted by a user to satisfy a given information need is called *(logical) session* [Baeza-Yates 2007]. Within a session, a transition (q, q') from a query q to a more specific query q' is called *specialization* [Boldi et al. 2009].

Following Capannini et al. [2011], we interpret the specializations of a given query as possible subtopics (intents) of that query. We represent the log sample that we use in this work by means of a query-flow graph [Boldi et al. 2008, 2009], which we can use to segment the query log into logical sessions in order to extract query specializations. Clearly, mining a query log does not ensure to retrieve all possible meanings of any given query. However, given the large size of the datasets, we can expect to find the most popular specializations [Capannini et al. 2011].

We use the relative frequencies of the specializations originating from a given query to compute the probability of each query intent. More precisely, given a query q and the set $\mathcal{S}_q = \{q'_1, q'_2 \ldots, q'_n\}$ of its specializations, we estimate the probability $\mathcal{P}(q, q'_i)$ of a transition (q, q'_i) as the ratio $f(q, q'_i) / \sum_{q'_j \in \mathcal{S}_q} f(q, q'_j)$ between the frequency of transition (q, q'_i) and the sum of the frequencies of all the possible specializations originating from q. In this setting, we obtain meaningful weights for the diversification task by adopting the following definition (inspired by the one introduced by Capannini et al. [2011]) for the *utility score* of a document u appearing in the result list of a query q:

$$w_q(u) = U(u|q) = \sum_{q' \in \mathcal{S}_q} \mathcal{P}(q, q') \sum_{u' \in \mathcal{R}_{q'}} \frac{CosSim(u, u')}{Rank_{u' \in \mathcal{R}_{q'}}}. \tag{3}$$

The overall weight of a document u for a query q is obtained by combining the relevance of u for q with the sum of the values of its utility score for every specialization q' of q, each weighted by the probability of that transition. The relevance of u for a specialization q' is computed as the sum, over all the documents u' appearing in the result list of q', of the cosine similarity between u and u', divided by the rank of u' in the result list of q'. The cosine similarities were computed on snippets extracted from the documents.

3.2. Setting the Weights

An important decision for a caching system that is based on our approach is how to set the weights $w_q(u)$ and the threshold W. As in every caching system, such decisions depend on the user requirements as well as on the available space that can be devoted to caching. For instance, if required to have at least 100 pages per query, we can set the values of $w_q(u)$ and W to obtain about 100 pages, or overestimate and set a bound on when a query is covered, as we do in the algorithm that we present in Section 4.2.

The exact values $w_q(u)$ also depend on the application and note that by setting the values appropriately we can optimize different objectives. To demonstrate the versatility of our approach we have proposed two different goals, ranking and diversification. But there are several other considerations that can be taken into account. For example, for different queries we might give more or less weight to the highest ranked results (our model and results hold for different weights $w_q(u)$ and threshold W_q for each query). This would be required for navigational queries in the web, where it is unacceptable not to present the top result(s) for a query, so we could set, for example, $w_q(u) = (1-\epsilon)W/k$ (for some small value of ϵ) for the top-k documents that we believe that are necessary to be returned, and a much smaller weight to other related documents.

4. ALGORITHMS AND ANALYSIS

The optimization problem defined by Problem 3.1 is a stochastic generalization of the set-multicover problem, in which elements correspond to queries and documents correspond to sets. In particular, we consider a stochastic setting that resembles the one by Grandoni et al. [2008], who studied the problem of *stochastic universal set cover*. Differently from the traditional set-cover problem, we need to define a fixed mapping from elements to covering sets without knowledge of the elements to cover, because we do not have a priori knowledge of the queries that will be submitted. Also, we are considering an extension of the above problem in which sets have associated element-dependent weights. We mention here that in a deterministic setting we cannot prove strong, meaningful results (as shown in [Jia et al. 2005]): the worst-case bounds there are very loose and do not provide a lot of insight; for more discussion refer to Section 4.3. Instead, we prove that knowing the query distribution suffices to provide algorithms with logarithmic (expected) approximations. In particular, in Section 4.1 we present a simple and efficient greedy algorithm for the QUERY MULTICOVER(t) problem and we prove that it achieves logarithmic approximation ratio under some reasonable assumptions. Note that the proof in [Grandoni et al. 2008] for the set-cover problem cannot be applied to our setting and that we need more sophisticated arguments to prove that our algorithm achieves a good approximation.

In the next sections we present our main algorithm and its analysis.

4.1. Stochastic Query Multicover

Algorithm 1 is called `Greedy Multicover` (`GM` for brevity). It is the straightforward greedy approach: in each iteration it selects the document that covers the largest total weight among the uncovered queries. We note that, even though the algorithm is simple, the analysis is nontrivial in our stochastic setting.

Let us now proceed with the analysis of the performance of the algorithm. For the sake of presentation, we assume that for each query q, the ith heaviest document has weight w_i, independently of q, as in the case that weights are based on the ranking. Nevertheless our results hold for a general weight structure with minimal modifications, mostly to the notation. Incidentally, notice that the same document might have different weights for different queries. Define ℓ to be the minimum integer such that $\sum_{i=1}^{\ell} w_i \geq W$, and note that we have $\ell \leq W/w_\ell$, because $w_\ell \leq w_i$, for $i \leq \ell$. Consider a sequence of t queries that are sampled from the distribution $\mathcal{Q}$. For a sequence $\omega = (q_1, \ldots, q_t)$, where the q_is are independently and identically distributed samples from $\mathcal{Q}$, let $C^{\mathrm{opt}}(\omega)$ be the optimal cost, that is, the minimum number of documents that can cover all the queries in ω, and let $\bar{C}^{\mathrm{opt}} = \mathbf{E}[C^{\mathrm{opt}}(\omega)]$ be the expected cost. Similarly, define $C(\omega)$ and $\bar{C}$ as the cost and the expected cost, respectively, induced by the `Greedy Multicover` algorithm. With respect to this setting, we now give our main theorem:

THEOREM 4.1. *For any sequence of t queries the mapping created by the `Greedy Multicover` algorithm satisfies*

$$\bar{C} = O\left(\frac{W}{w_\ell} \ln mn\right) \cdot \bar{C}^{\mathrm{opt}}.$$

Algorithm 1: Greedy Multicover Algorithm.

Function `Greedy Multicover (GM)`
/ If there exists an uncovered query, find the most cost-effective doc */*
while $Q \neq \emptyset$ **do**
  doc $\leftarrow$ mostCostEffectiveDoc();
  for $q \in$ queries(doc) **do**
    results$(q) \leftarrow$ *results*$(q) \cup \{doc\}$;
    if $\sum_{u \in \text{results}(q)} w_q(u) \geq W$ **then**
      $Q \leftarrow Q \setminus \{q\}$;
    end
  end
end

Function mostCostEffectiveDoc()
/ The most cost-effective doc is the one having the highest total weighted query coverage */*
for $u \in U$ **do**
  costEffectiveness$(u) \leftarrow \sum_{\{u:u \in results(q), q \in Q\}} f(q) \cdot w_q(u)$;
  if costEffectiveness$(u) >$ costEffectiveness(doc) **then**
    doc $\leftarrow u$;
  end
end
return *doc*

The proof of the theorem is rather technical and the interested reader can find it in Appendix A. It is heavily based on the following lemma, which, informally, states that the greedy algorithm is able to find a relatively small set of documents that covers most of the queries. We mention it because it will be useful subsequently given that it provides intuition about the greedy algorithm.

LEMMA 4.2. *Algorithm* `GM` *needs at most*

$$97\frac{W}{w_\ell}\bar{C}^{\text{opt}} \ln(nW/w_\ell)$$

documents to cover with weight W *all but* $8\frac{n}{t}\bar{C}^{\text{opt}} \ln mn$ *queries from* Q.

4.2. Fitting More Queries in the Cache

The approach that we have presented tries to achieve a desired rate of relevance by selecting documents, so that each query is covered for an overall weight of at least W. The problem with this strategy is that some queries might be covered by a large number of documents with small weights.

Often in information retrieval scenarios—and certainly in the applications that we mention in the introduction—we are interested in the precision at n: the number of documents among the top-n documents retrieved that are relevant to the query; the results in the lower ranks are of little relevance (note that this is not the case for measures such as mean average precision (MAP), which depend on the entire set of results). In such cases it is of limited value to store documents in the cache that are not among the top-n. Instead, if for a query we have stored a given number of relevant documents in the cache, we can consider the query covered even if these documents have a total weight less than the threshold W.

In light of this observation, we propose Algorithm 2, which is a modified version of the `Greedy Multicover` algorithm. We call it `Cardinality-Bounded Greedy Multicover` (`Card.GM` for brevity). At each step it selects the document with the highest total weight, with the difference that a query is considered completely covered if it has at least k relevant documents. In Section 5 we confirm that in practice this method is superior in the cache usage when precision-at-n is the measure of interest.

Algorithm 2: Cardinality-Bounded Greedy Multicover Algorithm.

Function `Cardinality-Bounded Greedy Multicover(Card.GM)`
/ If there exists an uncovered query, find the most cost-effective doc */*
while ***exists*** q ***s.t.*** $|\text{results}(q)| < k$ **do**
  doc $\leftarrow$ `mostCostEffectiveDoc();`
  for $q \in$ queries(doc) **do**
    results$(q) \leftarrow$ *results*$(q) \cup \{doc\}$;
  end
end

Function mostCostEffectiveDoc()
/ The most cost-effective doc is the one having the highest total weighted query coverage */*
for $u \in U$ **do**
  costEffectiveness$(u) \leftarrow \sum_{\{u:u \in results(q), q \in Q\}} f(q) \cdot w_q(u)$;
  if costEffectiveness$(u) >$ costEffectiveness(doc) **then**
    doc $\leftarrow u$;
  end
end
return *doc*

4.3. Discussion

In this section we discuss the knowledge obtained by our theoretical approach, and we leverage it to create a practical algorithm.

To comment on our model, the stochastic model is not only natural, but it is also necessary to obtain reasonable approximation guarantees; in fact, it is possible to prove a lower bound $\Omega(\sqrt{n})$ if no stochastic assumptions on the user behavior are made [Jia et al. 2005] and if we consider the worst case scenario (e.g., all users performing only uncommon and different queries), which sheds little light on the effectiveness of our approach in regular use. The downside is that the analysis of the stochastic case is technically demanding; for example the proof of Grandoni et al. [2008] (who consider the stochastic universal set cover, that is, when $w_q(u) = W$), which is already technically involved, cannot be applied to our case.

Another issue arises since the distribution $\mathcal{Q}$ is not known in advance. Assuming the knowledge of $\mathcal{Q}$ is a common assumption behind query-log mining. On the other hand, we show in our experiments that if sampling is performed in time windows that are similar to those in which the cache (or summary, more generally) is going to be used and not far in the past, the method is effective. In particular, we use the knowledge from a given period (it can be a day, or a given time period of a day) to obtain an estimate of the distribution and apply the algorithm for the next period (the same period next day, the same period next week, etc.). If the distribution changes over time, we expect that most of the queries will have similar probability to appear. Indeed, we study the overlap between queries in consecutive time periods, and, as expected, there is an overlap between queries with high frequency, whereas, queries in the tail of the distribution essentially do not overlap. This is one of the reasons that to learn the distribution we use only the previous time period, instead of accumulating queries over a longer period.

A careful examination of the proof also provides insight into the underlying picture. In particular, Lemma **??** describes important aspects of the problem structure, showing how to apply the greedy algorithm even when the cache size is limited. At a high level, the lemma states that (1) there exists a set of documents that covers a large fraction of the queries with weight W and (2) the greedy algorithm is able to recover it. Thus, even if the cache size is limited, we are able to apply the greedy algorithm and populate the cache with those documents, using prefetching. Note though that the proportion of queries that are not covered according to our lemma depends on the value of the optimal solution (for t requests), $\bar{C}^{\text{opt}}$. Intuitively, if the ratio $\bar{C}^{\text{opt}}/t$ is small, this

implies that there is a relatively small set of documents that cover many queries. This is because of overlap between the results lists of different queries, as well as because of the skewness of the query distribution. In practice, both of these situations are true: the result sets of web search queries such as "`hotel rome`," "`cheap hotel rome`," "`rome accommodation`," and so on, are expected to be highly overlapping, and similar is the situation in other settings where we can apply query-covering ideas (or at least where query covering is suitable). Also it is a well documented fact that query frequencies follow power-law distributions (see for example [Gan and Suel 2009; Lempel and Moran 2003]). Thus we expect the ratio $\bar{C}^{\text{opt}}/t$ to be small enough and thus the greedy algorithm to be able to cover many queries that will appear in the future by inserting documents in the cache. Our experimental findings, reported in Section 5, confirm the intuition obtained from our theoretical analysis.

5. EXPERIMENTAL RESULTS

In this section we present our experimental results. We start by describing the datasets and then we present the experiments. We consider two important criteria in document search: ranking relevance and diversification. We remark that our aim is not to propose a novel diversification strategy, but rather to show that our approach allows to easily design strategies that are sensitive to different optimization criteria, and the diversification of cached documents is just one—albeit important—example.

We start by describing the dataset collections that we created and used for our study.

In Section 5.2 we compare the performance of our algorithms with the standard techniques of top-k in terms of recall, efficiency in the use of the available cache, and in terms of precision. First we measure how many cache hits (recall) we attain for different caching strategies as we vary the number of queries that we cover in our training set. To measure cache efficiency, we observe how large of a cache we need to cover a given amount of queries from the training set, and what number of cache hits we have as we vary the available cache size. Finally, given that by construction our technique does not return all the top-k documents for a given query, we measure the precision at different levels, allowing us to evaluate the quality of the result page compared to not performing caching.

In Section 5.3 we answer the following question: "How well does the query-covering approach work together with a dynamic cache?" We implemented a static–dynamic caching technique available in the literature, in which we replaced the static-caching part with our query-covering technique. We show that the synergy is very effective and efficient.

We claim that the query covering is a very general technique. As an example to validate our claim, in Section 5.4 we attempt to show how caching can take into account diversification. To this end, we use some known techniques from the literature and we set the weights in such a way that the documents in the cache are diverse for the input queries.

We close the experimental section by reporting some numbers for the time required to apply our technique.

5.1. Datasets

We assessed the performance of our algorithm using queries from the *AOL* query-log dataset and documents that result from these queries. For the experiments on diversification we followed the guidelines of the *diversity task* of *TREC*.

Ranking. The query set that we used to evaluate the performance of our algorithms was extracted from the *AOL* query-log dataset [Pass et al. 2006]. This consists of about $36M$ query records, amounting to about $20M$ queries submitted by $650K$ users over a period of three months (from March to May 2006). The records are sorted by anonymous user ID and, for each ID, the queries are ordered by their submission times. Each entry in the dataset contains: the anonymous user ID (*AnonID*), the keywords submitted by the user (*Query*), the timestamp at which the search query was submitted (*QueryTime*), and the rank and the url of the clicked result (*ItemRank* and *ClickURL*). The

Table II. Examples of most frequent queries.

Wednesday 03/01/2006		**Thursday 03/02/2006**		**Friday 03/03/2006**		**Saturday 03/04/2006**	
Query	Freq	Query	Freq	Query	Freq	Query	Freq
google	2331	google	2428	google	1954	google	2403
ebay	1193	ebay	1132	ebay	1038	ebay	1196
yahoo	1015	yahoo	956	mapquest	867	yahoo	920
yahoo.com	853	yahoo.com	787	yahoo	853	mapquest	918
mapquest	804	mapquest	758	yahoo.com	744	yahoo.com	745
google.com	680	google.com	708	google.com	570	google.com	704
myspace.com	535	myspace.com	502	myspace.com	537	myspace.com	556
weather	523	intenet	444	internet	411	internet	466
internet	486	american idol	362	www.google.com	328	www.google.com	401
www.google.com	380	www.yahoo.com	358	myspace	328	www.yahoo.com	365

last two entries represent a click-through event, and they are present only if the user clicked on a search result. The query that preceded the click is always included, so if a user clicked on more than one result in the list returned from a query, there will be more lines for that query, one line for each click.

In our experiments, we used the *Query* and the *QueryTime* fields. In particular, we globally sorted query records by time and then took the queries submitted during an interval of four consecutive days (from March 1 to March 4, 2006). For this period, we observed about $1M$ queries, out of which 66% are distinct. This percentage is higher than the fraction of distinct queries over the entire period covered by the query log (around 50%), and it is mainly caused by high variance in the frequency of one-time and low-frequency queries.

We also observed an overlap among frequent queries of consecutive days, and we present a sample of the most frequent queries in Table II.

Given a query, we retrieved its first $\ell = 30$ results using the *Yahoo BOSS Search API*. Here the documents that appear in the result list of a query are considered relevant to the query. The average number of documents relevant to the queries submitted over a whole day is about $7M$, out of which about $4.3M$ are distinct; for two consecutive days the overlap of distinct documents is around $600K$.

Diversification. We assessed the effectiveness of our methods in diversifying documents using *ClueWeb-B*, the subset of the *TREC ClueWeb09*[4] dataset collection used in the *TREC 2009 Web track's diversity task* [Clarke et al. 2009]. The subset has a total of 50 million English Web documents. A total of 50 topics were available for this task. Each topic includes from 3 to 8 subtopics manually identified by *TREC* assessors, with relevance judgements provided at subtopic level. An example of topic and corresponding subtopics is shown in Table III.

5.2. Ranking Experiments.

For our experiments, the greedy multicover algorithm learns the distribution of queries by observing the stream of queries submitted over a suitable period of time. We can use the query log of a given day to cache documents and to serve queries submitted in the next day, or even in the same day of the next week, the underlying hypothesis being that the distributions of queries submitted at periodic (and not too distant) intervals will be similar. We denote the set of queries used in one time interval to learn the distribution and populate the cache by Q_{training} (*training set*), whereas the set of queries submitted in the next time interval is exploited to assess the performance of our algorithm and it is denoted by Q_{test} (*test set*). We used the first four days of March 2006 (from March 1^{st} (Wednesday) to March 4^{th} (Saturday)), so that the query log for March i was used to learn the distribution and populate the cache to serve queries submitted on March $i + 1$ ($i = 1, 2, 3$). We observed that the results achieved for a pair of consecutive days are similar to each other, so to obtain a succinct and at the same time significant set of values we decided to compute the average of all statistics over

[4]http://www.lemurproject.org/clueweb09.php/

Table III. TREC2009 Web track: Example of topic and corresponding subtopics.

```
<topic number="1" type="faceted">
    <query>obama family tree</query>
    <description>
        Find information on President Barack Obama's family history, including genealogy,
        national origins, places and dates of birth, etc.
    </description>
    <subtopic number="1" type="nav">
        Find the TIME magazine photo essay "Barack Obama's Family Tree".
    </subtopic>
    <subtopic number="2" type="inf">
        Where did Barack Obama's parents and grandparents come from?
    </subtopic>
    <subtopic number="3" type="inf">
        Find biographical information on Barack Obama's mother.
    </subtopic>
</topic>
```

three pairs of consecutive days. Recall that, given a query of the training set, we retrieved its first 30 results with *Yahoo BOSS*.

We implemented the two variants of the greedy algorithm presented in Section 4, `Greedy Multicover` (`GM`) and `Cardinality-Bounded Greedy Multicover` (`Card.GM`). The algorithm `GM` is parametrized by (x, W) and the algorithm `Card.GM` is parametrized by (x, k), where:

— x specifies the percentage of the queries of the training set that are selected to be covered by inserting documents in the cache.
— W is the coverage threshold. The query q of the training set is considered covered if the algorithm has selected a set of documents for which the sum of their weights is at least equal to W.
— k is the maximum number of documents that we want to retrieve from the cache (e.g., $k = 10$, that is, the number of documents appearing in the first page of results). The query q of the training set is considered covered if the algorithm has selected at least k documents in the result list of q.

To assign weights to documents for a given query we use Equation (2). There is a freedom on the logarithm base b that one can use, and later on we will report results obtained with $b = 10$, which turned out to trade off well recall, cache size, and precision. As we mention in Section 3.1, another natural approach is to omit the logarithm. However, we found that in such case the weights of the top documents dominate, resulting in a bad usage of cache space.

We compare the performance of the greedy approach (in its different versions) with the simple approach that selects the first k results for the most frequent queries.

More precisely, we tested the following algorithms:

— `Greedy Multicover` (`GM`): This algorithm is reported in Section 4.1 (see Algorithm 1). Weights are computed by using the weight definition given by Equation (2). The algorithm selects documents relevant to uncovered queries preferring the high-weighted documents. The algorithm takes as input parameters the percentage x of queries from training set that we want to cover with documents, and the threshold W. We remind the reader that given a threshold W, the query $q \in Q_{\text{test}}$ is considered covered if the sum of the weights of its documents is at least equal to W.
— `Binary-Relevance Greedy Multicover` (`Bin.GM`): This algorithm is similar to the previous one, with the difference that it uses a binary definition of weights. Formally, the query-document weight is 1 if the document appears in the result list of the query, otherwise it is 0. The query is covered if we have selected at least a minimum number of documents. Hence, we can assume that the query is covered if it has at least k cached documents, and the algorithm is parametrized by (x, k).

— `Cardinality-Bounded Greedy Multicover` (`Card.GM`): This is the algorithm of Section 4.2 (see Algorithm 2). The algorithm is parametrized by (x, k), and the weights are computed by using Equation (2). The algorithm selects the high-weight documents relevant to uncovered queries, but, differently from `GM`, the query $q \in Q_{\text{test}}$ is considered covered if there are at least k documents relevant to it.
— `Top-`k: This simple approach orders the queries by frequency, then for each query it selects the top-k documents. This algorithm is parametrized by (x, k). The ideas behind `Top-`k is commonly applied in the literature for static caching [Fagni et al. 2006; Xie and O'Hallaron 2002]. Intuitively, this approach makes inefficient use of the cache (because it ignores the query-document structure), but it can provide an upper bound on the coverage and a strong benchmark on the precision that can be achieved for a given fraction x of queries covered in the training set.

We evaluate our algorithms using three performance indices:

— `recall`: this is the percentage of covered queries in the test set. We also considered a weaker notion of recall, which gives partial credit if the sum of weights is less than W or the number of documents is fewer than k; qualitatively the results are the same. Note that when computing the average recall, each query is considered multiple times if it appears multiple times in the test set.
— `dim_cache`: this is the number of cached documents and it determines the cache size. We are interested in: (1) how many documents are cached by each algorithm, (2) the increase in memory as the coverage degree and the percentage of covered queries in Q_{training} increases, and (3) the behavior of `recall` as the cache size changes for the different algorithms.
— `p@n`: we evaluate the performance of the various algorithms taking the ranking into account. As a measurement we use the *precision-at-n* (`p@n`). For a given query in the test set, this is the proportion of the top-n results of the query that are found in the cache. We then aggregate by averaging over all the covered queries in the test set.

Algorithm Comparison. We performed experiments varying parameters W (for `GM`) and k (for `Bin.GM`, `Card.GM`, and `Top-`k). Changing the values of one of these parameters may improve performance with respect to some metric but worsen it with respect to the others. In this paragraph we compare the results for values of parameters that strike the best tradeoff among the performance indicators we consider, as this allows fairer comparison of the different approaches. For the `Bin.GM`, `Card.GM`, and `Top-`k algorithms, parametrized by k, we present results obtained with $k = 10$. For the `GM` algorithm, which is parametrized by W, we tried a variety of different thresholds W and we found that $W = 10.0$ provides a behavior that is satisfactory with respect to recall and cache usage and comparable with the coverage definition of the other approaches. For instance, with this value of W, about 75% of the documents are covered by at most 10 queries, even though half of the documents are covered by 5 documents or less and a quarter of them by more than 10, in some cases more than 20.

We start by comparing the four approaches with respect to the coverage that they are able to achieve. In Figure 1 we present the `recall`. For each algorithm we fix the percentage x of queries that are covered in the training set, and we measure the percentage of queries from test set that are completely served by the documents in the cache. As expected, `Top-`k has the best performance, however `Bin.GM` and `Card.GM` are following closely, whereas `GM` performs worse, which we attribute to the fact that the selected documents are fewer (see Figure 2).

To measure the cache size usage we compare in Figure 2 the `dim_cache` varying the percentage x of queries that are covered in the training set. Not surprisingly, `Top-`k has to pay for its good performance: the space required grows linearly with the number of covered queries in the training set. The dimension of the cache is optimized when we apply the other approaches. For `Bin.GM` and `Card.GM` the space utilization is limited when the percentage of covered queries from training set is up to 40%, after which we observe a steeper increase. This is in line with the considerations of Section 4.3 that the greedy approach is able to find a small number of sets that cover a good percentage of the queries. This result, combined with the fact that after 40% the recall does not