Optimizing Content Freshness of Relations Extracted From the Web Using Keyword Search

Mohan Yang [1,2]* Haixun Wang [2] Lipyeow Lim [3] Min Wang [4]
[1]Shanghai Jiao Tong University, mh.yang.sjtu@gmail.com
[2]Microsoft Research Asia, haixunw@microsoft.com
[3]University of Hawai‘i at Mānoa, lipyeow@hawaii.edu
[4]HP Labs China, min.wang6@hp.com

ABSTRACT

An increasing number of applications operate on data obtained from the Web. These applications typically maintain local copies of the web data to avoid network latency in data accesses. As the data on the Web evolves, it is critical that the local copy be kept up-to-date. Data freshness is one of the most important data quality issues, and has been extensively studied for various applications including web crawling. However, web crawling is focused on obtaining as many raw web pages as possible. Our applications, on the other hand, are interested in specific content from specific data sources. Knowing the content or the semantics of the data enables us to differentiate data items based on their importance and volatility, which are key factors that impact the design of the data synchronization strategy. In this work, we formulate the concept of content freshness, and present a novel approach that maintains content freshness with least amount of web communication. Specifically, we assume data is accessible through a general keyword search interface, and we form keyword queries based on their selectivity, as well their contribution to content freshness of the local copy. Experiments show the effectiveness of our approach compared with several naive methods for keeping data fresh.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications

General Terms

Performance

Keywords

content freshness, synchronization strategy, web crawling

*Work done while the author is at Microsoft Research Asia.

1. INTRODUCTION

Today, an increasing number of applications operate on data from the Web. Mashup applications are particularly good examples of such applications: they combine data from multiple external (web) sources to create a new service. In order to provide reliable, high quality, value added services, data availability and freshness are of paramount importance. The Internet provides no guarantees on connectivity and data availability; hence data availability is often achieved by maintaining a local copy of the data of interest. These applications are then faced with the problem of keeping the data in the local copy fresh.

In this paper, we study the problem of refreshing the local data maintained by these applications. The problem is particularly challenging because the Internet that sits between the local data and the data source is unreliable, has relatively high latency, the change characteristics of the data at its source is generally not observable, and most web data sources only support very limited data retrieval APIs, i.e., mainly keyword search.

Keeping a local data set fresh has been studied previously in the context of web crawling. However, the problem we face is quite different. First, the goal of web crawling is to maintain a local copy of raw web pages from millions of web sites. Commercial search engines are built upon web crawling techniques. It can take weeks for a commercial search engine to completely refresh its entire repository. For the purpose of web search, it is often acceptable that a local copy of a web page is not updated for days or even weeks in the search engine. It is true that stale pages may reduce the relevancy of web search results, but the problem is not as serious as in our case, where stale content may lead to wrong answers in applications that provide specific services (e.g., checking the availability of a product, as in ticket reservations). Second, web crawling handles *raw* web pages and is agnostic to the content of the web pages. In contrast, the applications we are concerned with understand the content or the semantics of the data that need to be kept fresh in the local copy. While such applications are much more sensitive to the staleness of the data than applications such as search engines, it also creates an opportunity for developing new mechanisms for keeping the content (not just the raw web pages) fresh. For example, knowing which data items are more volatile or more important enables the refresh algorithm to raise their priorities for synchronization. Furthermore, we can group data items together according to application semantics, and use one Web access to refresh them.

1.1 Applications

Before we formally describe the problem and the challenges, let us consider some representative application scenarios.

- **Online Specialty Stores.** Suppose one manages a specialty store on the Web. Instead of selling his own stock of merchandise, he links products on his Web pages to major e-commerce sites such as Amazon.com, Buy.com, ebay.com, etc., and earns commission from them. Usually, a specialty store sells products in focused categories, which can be narrow or broad. For example, one can build an online drugstore that has the most comprehensive coverage of medications by pulling data from other general purpose stores, or one can specialize in books on closely related topics, or furniture of particular styles. To maintain such a store, one needs to synchronize with the data source (e.g., Amazon.com) frequently to keep information such as price and availability up-to-date.
- **Paper Citations.** Citations are a good measurement for scientific impact, and more and more institutes take citations into consideration when they evaluate job applicants or award candidates. Suppose a web application maintains multiple lists of publications, and publications in each list are on a particular topic (e.g., a reading list of web crawling literature), authored by a particular researcher, or by a group of researchers (e.g., faculties in Stanford's CS department). Suppose among everything else, the application obtains the citation count of each publication from Google Scholar. The challenge is to maintain the freshness of this materialized view.

1.2 Challenges

Many applications that rely on third party data sources face similar challenges as the two examples given above. The architectures of these applications have many things in common: to ensure data availability, a web application maintains one or multiple data sets, each materializing certain content from one or more external sources on the Web. In order to serve its clients using its materialized content, the application must poll the data sources regularly in order to ensure the freshness of the materialized content. The challenges lie in how to poll the sources.

- The first challenge is that the content we need to materialize locally is often big and dynamic. Take paper citations as an example. Its goal is to maintain publication lists for multiple researchers, for multiple institutes, and on multiple topics in order to provide more useful services such as topic analysis, etc. However, the content is dynamic since new papers and new citations are being added constantly. Consequently, the application needs to devise intelligent methods for materializing the large and dynamic content. Similar challenges exist for online specialty stores, as the availability and prices of products of interest also change constantly.
- The second challenge is the lack of efficient data access methods on the Web. Ideally, we want to query entities (e.g., publications and products) we are interested in as if they are structured relations. Thus, we would like to view the data source as a database, so that we can issue SQL-like queries against it. However, the data sources are either not structured or if they are, the schema of the data is usually not published. There are many reasons for not doing so, among which is the cost issue: for example, during schema evolution, APIs must be changed which may cause interruption of service. The most common interface for querying the external data source is through keyword search, which is the case for searching papers on Google Scholar and searching for products on Amazon.com.
- The third challenge is access restrictions. Most web sites have heavy restrictions against crawling. Take Google Scholar for example. Google will direct users that generate frequent polling to a CAPTCHA test to ensure that the requests are not coming from a robot. A naive method is to limit the number of queries sent to the data source within a given period of time (e.g., limit it to say 10 queries in any given 5 minute period). However, this approach will limit the functionality of the services we want to provide, especially when the data sets are large.

Some of these issues are also faced by the traditional web crawling problem (usually carried out by search engines), but our problem is unique in several aspects. First, our goal is to obtain specific information for a specific set of entities (papers, products, etc). The goal of Web crawling is to obtain as many raw pages as possible. Typically, the applications we consider has one or several targeted data sources on the Web, while web crawlers employed by commercial search engines scan millions of hosts. Recently, the topic of deep web crawling has attracted much interest. Just like general purpose web crawling, its goal is also to obtain as much information from the database beneath the surface web as possible.

Second, we are concerned about content freshness. For instance, a significant change of a product's price or a paper's citation justifies high priority of refreshing the information about the product. Web crawling, on the other hand, is only concerned about whether a raw page has been updated. Thus, there is no difference from page to page, or the nature of change within a page.

Third, we have a more stringent requirement on data being fresh. For search engines, there are typically no correct answers when it comes to search results. Instead, search engines provide a list of results, hopefully ranked by their relevancy. It is acceptable that the crawled pages are a few days or a few weeks old. However, the applications we are concerned with must analyze (e.g. aggregate) the content to provide a new type of service. Thus, most of them require that the content (e.g., product availability and price) is accurate and up-to-date.

1.3 Our Approach

Our goal is to maintain the freshness of a set of entities by sending as few web requests as possible to the sites where such information is hosted. We assume that each data item has a unique ID assigned by the data source. For instance, each product on Amazon.com has a unique ID, and each paper hosted by Google Scholar has a unique ID. Furthermore, we assume we can form a web request using an item's ID to obtain the information about the item. However, accessing information by ID is not efficient: in order to refresh the entire repository of N items, we need to send N requests. Besides being inefficient, sending many requests within a short period of time is usually unacceptable due to web access restrictions (excessive accesses of a Web site such as Google Scholar will be rejected by the Web site).

In addition to ID-based data retrieval, data sources usually provide keyword based search. For each search, items that contain the keywords in the search are returned (unless the keywords are too popular, in which case, a subset of the items are returned). Thus, if the result of one search contains information about multiple entities we are interested in, then we save on the number of web accesses. This is possible because the entities we are concerned with are not a random set of entities. Instead, they usually have some commonality. For instance, publications in a list are either on a specific topic, or authored by one researcher, or by a group of researchers in the same institute, etc. For specialty stores, the seller focuses on grouping and presenting products in a better way than general retailers such as Amazon.com, which means products in a specialty store are often related. This offers opportunity to refresh multiple items in a single Web request.

In view of this, our approach focuses on finding the best set of keywords that can be used to refresh the data set in the most efficient way, i.e., with least number of searches. In the following, we discuss two factors that affect the keyword selection process.

- Keyword Selectivity. First, we should choose keywords that have high selectivity on the local data set (i.e., find the keywords that maximize the number of items in the local copy that contain the keywords). Second, we should choose keywords that have low selectivity on the server, such that most returned items are relevant to the local data set.
- Refreshing Priority. Not all items are the same. There are at least three factors that may impact the priority: time, volatility, and importance. Take papers in a publication list as an example. The time factor says that a paper that was refreshed a long time ago should have high priority to be updated. Second, different papers have different volatility. For example, highly cited papers, or papers on hot topics, or papers written by well known authors are more likely to attract citations, and are hence more volatile. Thus, they should have higher priority to be updated. Finally, certain items, e.g., items that many users are interested in, are more important, and should be kept fresh with higher priority.

In this paper, we introduce the concept of content freshness. Unlike previous work in maintaining data freshness, content freshness takes into consideration not only the time factor, but also factors related to items' volatility and importance. We then conceive a greedy approach: every time we select keywords in a way to maximize the increase of the content freshness.

1.4 Paper Organization

The rest of the paper is organized as follows. Section 2 compares our work with related work. Section 3 presents the mathematical formulation of the problem. Section 4 presents our greedy algorithm for the query generation and graph technique for querying order and frequency optimization. Section 5 presents the experimental results. The paper concludes in Section 6.

2. RELATED WORK

Cho et. al. [3] did some pioneering work on designing a crawling policy to maintain a web repository fresh. The goal of their work is to answer the following questions: "How often should we synchronize the copy to maintain, say, 80% of the copy up-to-date? How much fresher does the copy get if we synchronize it twice as often? In what order should data items be synchronized? For instance, would it be better to synchronize a data item more often when we believe that it changes more often than the other items?" Cho et. al. introduced a definition of data freshness, and proposed a crawling policy to increase the freshness. In particular, the crawling policy takes web page update patterns (e.g. Poisson) into consideration [3].

There are two major differences between Cho's work and ours. First, Cho et. al. maintain the freshness of a local data set that correspond to a set of URLs. Each URL potentially points to a different web site. Thus, each web request is for one URL and refreshes one element of the local data set: in other words, it is not possible to refresh multiple elements in the local copy by one Web request. In our work, the local data consists of a set of entities, which are extracted from a remote database hosted on a single Web site (e.g., paper citations based on Google Scholar) or a small number of Web sites (e.g., online specialty stores based on Amazon.com, buy.com, etc). Our goal is thus to maximize the number of entities retrieved using one query.

The second major difference is that Cho et. al.'s freshness model is based on the assumption that updates on a web site follow Poisson process. Because Cho et. al. maintains a set of URLs of different web sites, this is probably the best assumption they can make since they do not have any knowledge of the content of the data. In our case, we maintain a set of entities. Thus, we know the semantics of the data, so we can make more reasonable assumptions. For example, consider the publication/citation graph. We can assume that the citations follow a power-law distribution. In other words, a highly cited paper is more likely to attract more citations, and hence should be updated more frequently. The same phenomenon occurs in online specialty stores. A popular item (highly ranked on sales list) is more volatile, that is, it is more likely to go through changes (ranking, price adjustment, availability, etc). Furthermore, a popular item, just like a highly cited paper, is more likely to be queried on the local site as well.

Sia et. al. [14] studied how the RSS aggregation services should monitor the data sources to retrieve new content quickly using minimal resources and to provide its subscribers with fast news alerts. The optimal resource allocation and retrieval scheduling policy enables the RSS aggregator to provide news alerts significantly faster than the previous approaches. Our work is different from theirs in the following two aspects. First, their requests are clustered based on data sources to begin with as users subscribe news by domains. In our work, one challenge is to cluster entities so that they can be covered by a set of queries. Second, their optimization is performed on the data source level while ours is performed on the query level. Their work cares about how much resource should be allocated to each data source and when should a specific data source be synchronized in the allocated time slot based on the posting pattern in the data source. Our work cares about in what order should queries be sent and how frequently should we send queries. There are also a lot of research work on aggregation services, including [4, 10]. However, these methods passively wait for new data to come in instead of actively pulling data from the data sources.

Deep Web crawling [11, 9, 1, 2, 17, 8, 6, 7, 16] is another area that is closely related to our work. Raghavan and Garcia-Molina [11] proposed a generic operational model of a deep web crawler. They mainly focused on the learning of hidden web query interfaces. Ntoulas et. al. [9] developed a greedy algorithm to automatically generate keyword queries. The algorithm tried to maximize the efficiency of every query, thus obtaining as many pages as possible with a relatively small number of queries. Wu et. al. [17] also proposed a similar keyword query selection technique for efficient crawling of structured web sources. Barbosa et. al. [1, 2] worked on a project called *DeepPeep* which gathered hidden web sources in different domains based on novel focused crawler techniques. Madhavan et. al. [8] described a system for surfacing deep web content, i.e., pre-computing submissions for each HTML form and adding the resulting HTML pages into a search engine index. Wang et. al. [16] developed a greedy algorithm to solve the weighted set covering problem targeting at deep web crawling. However, deep web crawling focus on the problem of discovering as many items as possible from the data source while our work mainly focuses on the scenario of maintaining a specific set of items. Thus, the crawling strategy in the above work does not apply, as we are not interested in the irrelevant items returned by the queries. Additionally, content-freshness or result-freshness is never considered in these works, it is an important dimension of deep web crawling and the main focus of our paper.

3. PROBLEM FORMULATION

We model a web data source as a server database. Without loss of generality, consider the case where the server database contains a single relational table. Note that the table is used as the database backend of a web application, although it has a schema, it does not support relational queries from the user. Instead, all queries must go through a web interface. Thus, we assume, as far as queries are concerned, (1) each row in the table is represented by a bag of words, and (2) the web interface only supports keyword queries on the table. Each query consists of a set of keywords, and can retrieve up to K tuples that contain all of the keywords in the query.

An application that keeps a local copy of the web data is modeled as a local database. The local copy contains a subset of the rows in the server database, and its content is kept up-to-date by querying the server database periodically using hopefully a small number of queries. Furthermore, the local database can only query or probe the server database via keyword queries. At each synchronization, the application needs to decide on a set of keyword queries to probe the server database in order to maximally refresh the content of its local copy. In most practical applications, the synchronization occurs at fixed intervals.

S	Server copy of the relation.
L	Local copy of the relation.
$k(q)$	Keywords associated with query q.
$w(r)$	The weight associated with tuple r in the local relation.
$R_S(k)$	The set of tuples in the server relation associated with keyword k.
$R_L(k)$	The set of tuples in the local relation associated with keyword k.
K_L	The set of all keywords associated with the local copy of the relation.

Table 1: Notations

There are two optimization objectives when choosing the set of keyword queries at each synchronization point.

1. the local relation should be kept as up-to-date, i.e., as similar to the current content on the server as possible, and
2. the number of keyword queries should be as small as possible.

3.1 Optimization Objective 1

A key element in the first objective is that we need a similarity measure between the local data and the data on the server. Let r_L be a tuple in the local database L. Let r_S be the corresponding tuple in the server database S. Assume we have a function $\delta(r_L, r_S)$ that measures the dissimilarity between r_L and r_S. However, not all tuples are equal. The freshness of some tuples is more important than others. To reflect the importance, we assign a weight $w(\cdot)$ to each tuple. Thus, the priority of a record is given by its weighted dissimilarity:

$$w(r_L) \cdot \delta(r_L, r_S)$$

In other words, the higher the weight of a tuple, and the bigger the difference between the server and the local copy, the higher the priority to refresh the tuple. Then, the content freshness of the entire local relation with respect to the server relation can be computed as:

$$\delta(L, S) = \frac{1}{|L|} \sum_{r_L \in L} w(r_L) \cdot \delta(r_L, r_S)$$

Note that a content freshness of zero means that the local database is perfectly in-sync with the source. The larger the content freshness, the less "fresh" the local content.

Both the dissimilarity function and the weight function are application dependent. For example, for the paper citation application, the dissimilarity function can be defined as the absolute value of the difference between the number of citations on the local and the server databases, i.e.,

$$\delta(r_L, r_S) = |r_L.\texttt{Num\_of\_citation} - r_S.\texttt{Num\_of\_citation}|$$

where `Num_of_citation` is a database column. Similarly, we can conceive a weight function which says the more citations a paper has, the more important the paper is, i.e.:

$$w(r_L) = r_L.\texttt{Num\_of\_citation}$$

In the same spirit, we can define the importance based on the popularity of its author, the relevance of its topic, or the prestige of the conference or journal where the paper appears, etc.

As we can see, the weighted dissimilarity is the key to the concept of content freshness, as it employs the semantics of the data to decide on the refreshing strategy. This also makes our problem different from web crawling, which is mainly concerned with whether a web page has been updated or not.

Given the weighted dissimilarity, we design a synchronizing strategy to keep the data set fresh. Intuitively, the local application measures the difference between the local tuples and the server tuples, and gives priority to refreshing those that have a big weighted difference when it synchronizes with the server. But there is a big catch: the local application does not know what the current r_S is. We assume r_S has changed since the last synchronization (otherwise $\delta(r_L, r_S)$ will be 0, which means there is no need to refresh it), but we do not know how much it has changed.

In our approach, we estimate r_S based on i) the content of r_L, ii) the volatility of r_L, and iii) the time elapsed since the last synchronization of r_L. In other words, we assume $\delta(r_L, r_S)$ can be estimated by $F(r_L; t)$, or simply

$$\delta(r_L, r_S) = F(r_L; t)$$

where t is the current time. In other words, we create an update model of the content. We consider r_L's volatility and previous synchronization information as features of r_L. Thus, from r_L and t we can derive r_S. This leads to our definition of content freshness.

DEFINITION 1. *The content freshness of the entire local relation at time t is defined as:*

$$F(L; t) = \frac{1}{|L|} \sum_{r_L \in L} w(r_L) \cdot F(r_L; t) \tag{1}$$

As an example, a simple deterministic update model for the citations can be the following. Let t_s be the time that r_L was last synchronized. Let ω be the parameter that reflects the volatility of r_L – for instance, assume the citation of a paper increases by 1 every ω days. Thus, we have

$$\begin{aligned} F(r_L; t) &= \delta(r_L, r_S) \\ &= |r_S.\texttt{Num\_of\_citation} - r_L.\texttt{Num\_of\_citation}| \\ &= (r_L.\texttt{Num\_of\_citation} + \frac{t - t_s}{\omega}) - \\ &\quad r_L.\texttt{Num\_of\_citation} \\ &= \frac{t - t_s}{\omega} \end{aligned}$$

Certainly, this is a very naïve update model. In Section 4, we introduce more realistic models where the change of the citation is modeled by a Poisson process with a rate parameter obtained from historical data.

3.2 Optimization Objective 2

The second objective seeks to minimize the number of necessary keyword queries for synchronization. This requirement is due to a limitation in web based keyword search: for each query, the web server only returns top K results that match the query. In other words, even if there exists a common keyword that matches every tuple in the server database, a query containing the keyword will not retrieve the entire database. More likely it returns K tuples that the local application has little interest in.

Thus, a "good" keyword query is a keyword query that ensures most of the returned K tuples are useful and important to the local application. By "useful and important", we mean they are part of the local copy and they have high weighted dissimilarity. Thus, obtaining these tuples through synchronization will greatly improve the content freshness of the local data. This requires the query to contain keywords that match as many important tuples in the local copy, and as few irrelevant tuples, as possible. To make this optimization decision, the local database needs to estimate the selectivity of keyword queries on the server database. That is, given a set of keywords, we need to estimate how many tuples in the server database contains these keywords. Then, we will avoid keywords that match a large number of tuples in the server database.

3.3 Problem Statement

We give our problem stalement as follows:

At each synchronization, find a set of queries Q such that after getting the results of Q, the content freshness of the local relation is maximally improved (that is, $F(L; t)$ is maximally reduced.)

A couple of constraints on queries are necessary to fully specify the problem: (1) only relevant keywords are to be used in the queries, $\forall q \in Q, k(q) \subseteq K_L$, (2) the keywords in the queries cover the local relation, $L \subseteq \cup_{q \in Q} R_L(k(q))$, (3) the number of queries $|Q|$ should be minimized, and (4) the number of tuples transmitted, $\sum_{q \in Q} |R_S(k(q))|$ should be minimized.

A Note on NP-Hardness. Consider a more constrained version of the problem where the weights of the dissimilarity measure between local and server relation are set to one. The problem then is to find the smallest number of queries to cover the whole local relation while the number of tuples transmitted is minimized. Let $\mathcal{Q}$ be the set of all possible queries (i.e., all possible combination of keywords in K_L), then we are trying to solve the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{q \in \mathcal{Q}} |R_S(k(q))| x_q \\ \text{subject to} \quad & \sum_{q : r \in R_L(k(q))} x_q \geq 1 \quad \forall r \in L \\ & x_q \in \{0, 1\} \quad \forall q \in \mathcal{Q} \end{aligned}$$

where x_q represents whether a query $q \in \mathcal{Q}$ is selected(1 for yes, 0 for no) when objective function is minimum. This is the *covering integer program* [15] which is a generalization of the set cover problem.

4. OUR APPROACH

Our approach to refresh the content of a local relation L with a server relation S is outlined in Algorithm 1. The REFRESHLOCAL algorithm contains an infinite loop of refresh events. Each refresh event, i.e., each iteration of the loop, generates a set of keyword queries using the GREEDYPROBES algorithm, uses the generated set of keyword queries to retrieve data from the server relation, updates the local relation using the retrieved data, and sleeps for τ time units. At this point, for ease of exposition, we assume that the results

Algorithm 1 REFRESHLOCAL(L, S)

```
Input: local relation $L$, information related to server relation $S$
Output: synchronized local relation $L$
1: loop
2:     $P \leftarrow$ GREEDYPROBES$(L, S)$
3:     $\mathcal{D} \leftarrow$ query $S$ using $P$
4:     update $L$ using results $\mathcal{D}$
5:     sleep for time interval $\tau$
6: end loop
```

Algorithm 2 GREEDYPROBES(L, S)

```
Input: local relation $L$, information related to server relation $S$ and statistics to compute $|R_S(q)|$ for query $q$
Output: $P$
1: $P \leftarrow \emptyset$
2: $R_{\text{notcovered}} \leftarrow L$
3: while not stopping condition do
4:     $\mathcal{K} \leftarrow$ find all keywords associated with $R_{\text{notcovered}}$
5:     Pick $q$ from powerset $\mathcal{P}(\mathcal{K})$ using greedy heuristic Eqn. (2)
6:     $P \leftarrow P \cup \{q\}$
7:     $R_{\text{notcovered}} \leftarrow R_{\text{notcovered}} - R_L(q)$
8: end while
9: return $P$
```

for all the keyword queries are retrieved instantaneously. In Section 4.3, we remove this assumption and analyze the effect of ordering the keyword queries for obtaining maximum freshness.

The essence of our solution lies in the GREEDYPROBES algorithm that directly addresses the problem statement of the previous section. Given the NP-hardness of the problem, GREEDYPROBES uses a greedy heuristic-based algorithm outlined in Algorithm 2 to find the set of keyword queries for each refresh event.

The GREEDPROBES algorithm starts with an initially empty set of query probes P and adds a query probe to this set using a greedy query efficiency heuristic (to be described in Section 4.1). The stopping condition of the `while`-loop can be application dependent and some examples are: (1) when some quota of keyword queries has been reached, or (2) when the current set of queries completely covers the local relation. A query probe can be a single keyword or a conjunction of keywords; hence, a query is picked from the powerset of all possible keywords associated with the local relation. Note that in practice, the power set is never materialized, because of its exponential complexity.

Next, we describe the query efficiency heuristic in detail. The rest of the section will discuss how relaxing the assumption that the set of query probes are executed instantaneously will affect the change of content freshness. The paper citation scenario will be used as a running example in our analysis.

4.1 Query Efficiency Heuristic

For a keyword query q, let $|R_L(q)|$ and $|R_S(q)|$ be the local coverage and server coverage of q respectively. To minimize the number of queries, the local coverage of every query should be as high as possible. On the other hand, the search result of most keyword search interfaces is partitioned into pages while each page covers only a limited number of results. So if a query has large server coverage, i.e., the results of this query is partitioned into many pages, then a lot of additional queries are needed to retrieve all the result pages from the server. To minimize the number of tuples transmitted, the server coverage of every query should be as low as possible. Intuitively, a good query should maximize the query efficiency as defined by

$$q := \arg\max_q \frac{|R_L(q)|}{|R_S(q)|}.$$

This is similar to the tf/idf measure in IR. The term frequency (tf) represents the number of times a term occurs in a document. The inverse document frequency (idf) is a measure of the general importance of the term. A high weight of tf/idf measure is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents.

As different tuples have different priorities to be refreshed, it is preferable to cover high priority tuples. Let $w(r)$ be the importance of tuple r, and let $F(r;t)$ be the dissimilarity between r and the current r on the server. Then we can change the local coverage into weighted local coverage, i.e., for a query q we have $\sum_{r\in R_L(q)} w(r)\cdot F(r;t)$ instead of $|R_L(q)|$, which is a special case where all $w(r)$ and all $F(r;t)$ are the same). Still, we want to maximize the efficiency

$$q := \arg\max_q \frac{\sum_{r\in R_L(q)} w(r)\cdot F(r;t)}{|R_S(q)|} \tag{2}$$

Estimating Server Coverage. In Equation (2), the numerator part can be directly calculated based on the local relation L and an update model which we will introduce in Section 4.2. But to compute the denominator part, we need information from the server. Usually, a keyword search interface will notify user about the total number of results related to a query on the result page. This is a way to obtain the value of $|R_S(q)|$. However, this method requires sending queries to the server for every possible value of q. The query generation may need the information of every possible combination of keywords. Such a huge amount of queries may be blocked by the keyword search interface. So we need some alternative methods to indirectly estimate the value of $|R_S(q)|$.

One possible approach is to use the technique developed by Li and Church [5], which estimates word associations (cooccurrences of words, i.e., the frequency that a set of words occur together). If we regard the server relation as a corpus, the keyword query can be seen as calculating the word association for words in a keyword query. Li and Church's method samples the inverted index of the corpus, and constructs a contingency table for this sample. Then, a straightforward method for deriving the contingency table for the entire population is to use scaling. Li and Church use a maximum likelihood estimator (MLE) by taking advantage of the margins (also known as document frequencies) to make the estimation more accurate [5]. However, in order to use Li and Church's method, we need to create the inverted index first, which means we need to crawl the entire server.

Another possible approach is to use some similar data sources to estimate the value of $|R_S(q)|$. Data sources from the same domain are similar both in terms of their attribute values and their attribute value distributions. Sometimes, downloadable data sources in the same domain are available. The downloaded data can act as a good estimation of our target server data. With the help of the downloaded data, the estimation can be rather simple. For example, if we need the $|R_S(q)|$ value for some computer science papers on Google Scholar, we can use the data from DBLP[1] as an estimation. DBLP includes more than one million articles on computer science. Let $|R_{DBLP}(q)|$ be the number of items related to query q in the DBLP data dump, then the value of $|R_S(q)|$ can be estimated as $|R_{DBLP}(q)| \cdot c$. c is a scaling factor representing the volume difference of these two data sources. c can be estimated by comparing the results number of several queries on these two data sources.

4.2 Poisson Analysis

In Section 4.1, we introduced the GREEDYPROBES algorithm to find a set of queries that cover the local dataset. Suppose the set of queries found is $P = \{q_1, q_2, \ldots, q_k\}$ and $R_L(q_1), R_L(q_2), \ldots, R_L(q_k)$ denote the corresponding results of the k queries.

In many practical applications, the number of queries, k, can be quite large. For instance, a website that keeps track of thousands of products sold on Amazon.com may need hundreds if not thousands of queries to synchronize its list. Previously, we have assumed that these k queries can be executed and results obtained from the server database instantaneously. When k is large, the assumption is not valid. Moreover, the server may not even allow such a large number of queries from a single client in a single refresh event. Thus, we need to "spread out" the queries over time and try to maximize the content freshness with less frequent queries to the server. One of the simplest models is to issue one query at fixed time intervals. Let I be the fixed time interval. At each time slot, we issue exactly one query, and thus we finish one iteration of the synchronization loop in the REFRESHLOCAL algorithm within a time span of $kI + \tau$ time units.

Two questions immediately arise with this model. First, what is the order of issuing the queries, and second, how does the fixed time interval I impact the content freshness of the dataset. To answer these two questions, we introduce a theoretical content freshness measure and a Poisson update model of the source data in order to perform the analysis for these two questions.

Definition of $F(e;t)$. Suppose we measure the content freshness of an item e at time t as follows.

$$F(e;t) = \begin{cases} 0 & \text{if } e \text{ is up-to-date at time } t \\ \sum_{t_i \in U} t - t_i & \text{otherwise} \end{cases}$$

where U is the set of time points at which an update occurs on the server (e changes value) since last synchronization. Then the content freshness of the local relation L at time t is

$$F(L;t) = \frac{1}{|L|} \sum_{e \in L} w(e) \cdot F(e;t) \quad (3)$$

Fig. 1 shows an example illustrating this definition of content freshness. The content freshness of e stays zero before the first updating. After that the content freshness keeps growing at a speed of one till the first synchronization. After the first synchronization, the content freshness drops to zero and remains zero till next updating. The content freshness keeps increasing from the third updating. Since no synchronization has occurred since the third updating, the content freshness keeps growing at the speed of two after the fourth updating and at the speed of three after the fifth updating. If the time of the third, fourth and fifth updating is 5, 6 and 7 respectively, then the content freshness of e at time 6, 7 and 8 is shown in Table 4.2.

[1]The DBLP Computer Science Bibliography: `http://www.informatik.uni-trier.de/~ley/db`

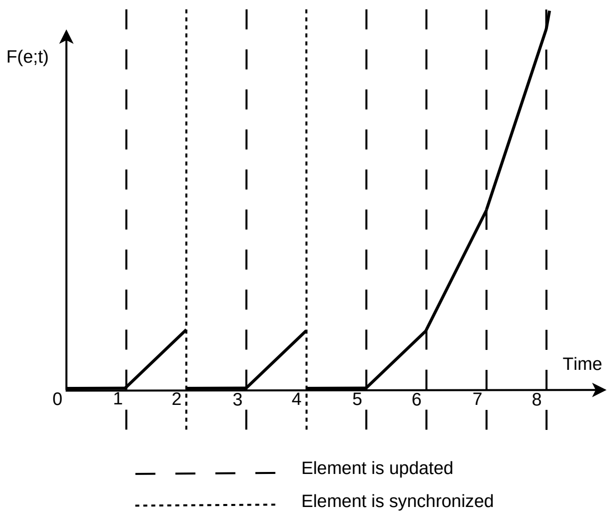


Figure 1: An Example of the Time Evolution of Content Freshness

time	content freshness
6	6 - 5 = 1
7	(7 - 5) + (7 - 6) = 3
8	(8 - 5) + (8 - 6) + (8 - 7) = 6

Table 2: $F(e;t)$ of e at time 6, 7 and 8

Poisson Update Model. Suppose we model the update characteristics of a source data element (eg. paper citation count) using a *Poisson process*[2]. For an item e, the parameter of the Poisson process is λ which represents the number of update events in one time unit. A Poisson process implies that update events do not occur simultaneously. In the paper citation scenario, this means that the citation count can only increase by one at each update event.

Let $N(t)$ be the number of update events within a time period of t. Then we have the following based on the Poisson assumption:

$$P(N(s+t) - N(s) = k) = \frac{(\lambda t)^k}{k!} e^{-\lambda t}$$

Next, we derive the expected content freshness at time t under this update model. Let the sequence of Poisson update

[2]Cho et. al. [3] verified the changes of Web pages followed a *Poisson process*. As the information of source data elements is extracted from Web pages, the update characteristics should also follow a *Poisson process*.

event times be $\tau_1, \tau_2, \cdots, \tau_k, \ldots$. At time x, we have

$$\begin{aligned} P(\tau_k = x) &= \lim_{\mathrm{d}t \to 0} \frac{P(N(x+\mathrm{d}t) \geq k, N(x) = k-1)}{\mathrm{d}t} \\ &= \lim_{\mathrm{d}t \to 0} \frac{1 - P(N(\mathrm{d}t) = 0)}{\mathrm{d}t} P(N(x) = k-1) \\ &= \lim_{\mathrm{d}t \to 0} \frac{1 - (1 - \lambda \mathrm{d}t + \mathcal{O}(\mathrm{d}t^2))}{\mathrm{d}t} \frac{(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \\ &= \lambda \frac{(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \end{aligned}$$

Hence, the contribution of each k-th update to the expectation of content freshness is,

$$\int_0^t (t-x)\lambda \frac{(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \mathrm{d}x$$

Adding up the above expression for $k = 1, 2, \cdots, \infty$ gives

$$\sum_{k=1}^{\infty} \int_0^t (t-x)\lambda \frac{(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \mathrm{d}x = \frac{1}{2}\lambda t^2$$

However, this expression does not guarantee the order $\tau_1 < \tau_2 < \cdots < \tau_k < \cdots$. So some unsatisfactory situations should be excluded from this value, i.e.,

$$E[F(e;t)] < \frac{1}{2}\lambda t^2$$

On the other hand, $E[F(e;t)]$ must be bigger than the expectation of τ_1, so

$$\begin{aligned} E[F(e;t)] &> \int_0^t (t-x)\lambda e^{-\lambda x} \mathrm{d}x \\ &= t\left(1 - \frac{1 - e^{-\lambda t}}{\lambda t}\right) \\ &\sim \frac{1}{2}\lambda t^2 \end{aligned}$$

Combining the upper and lower bound, we have

$$E[F(e;t)] \sim \frac{1}{2}\lambda t^2,$$

as the expression to represent the expectation of content freshness at time t since time 0.

4.3 Query Order Optimization

Let g_i be the group of data elements covered by issuing query q_i, i.e., we use g_i as a shorthand for $R_L(q_i)$. After issuing a query q_i, the elements in the group g_i are synchronized together. For our analysis, we assume $g_i \cap g_j = \emptyset, \forall i \neq j$ (if they are not we can artificially remove duplicates from the groups). Let x_i denote the time when g_i was last synchronized and x_i satisfies the constraints $x_i \in \{0, 1, 2, \cdots, k\}$, $i = 1, 2, \cdots, k$, and $x_i \neq x_j$ if $i \neq j$. Then, if we send one query in a time unit I, the expected content freshness of L at time $t = kI$ is

$$\begin{aligned} E[F(L;kI)] &= \frac{1}{|L|} \sum_{i=1}^{k} \sum_{e_j \in g_i} w(e_j) E[F(e_j; (k - x_i)I)] \\ &= \frac{1}{|L|} \sum_{i=1}^{k} \sum_{e_j \in g_i} \frac{1}{2} w(e_j) \lambda_j (k - x_i)^2 I^2 \end{aligned}$$

Let a_i be a shorthand for $\frac{1}{|L|} \sum_{e_j \in g_i} \frac{1}{2} w(e_j) \lambda_j I^2$, which for fixed I and local copy of relation L, a_i is a positive constant. We have

$$E[F(L;kI)] = \sum_{i=1}^{k} a_i (k - x_i)^2 \tag{4}$$

To achieve the smallest content freshness L, we want to minimize Eqn. (4).

Clearly, the order of the queries, which is embodied by the vector of x_i, makes an impact on the content freshness. In Table 3, we use an example to demonstrate the relationship between the ordering and the content freshness. Assume i) the local dataset maintains 4 items, $L = \{e_1, e_2, e_3, e_4\}$; ii) item has the same weight, that is, $w(e_i) = \{1, 1, 1, 1\}$; iii) each item is updated by a Poisson process, with parameter $(\lambda_i) = (1, 2, 3, 4)$, and iv) e_1 and e_2 are covered by query q_1, e_3 and e_4 are covered by query q_2.

Table 3: Content Freshness L for $k = 2$, $I = 1$, $(\lambda_i) = (1, 2, 3, 4)$, $w(e_i) = \{1, 1, 1, 1\}$

	$x_1 = 0$	$x_1 = 1$	$x_1 = 2$
$x_2 = 0$	5	3.875	3.5
$x_2 = 1$	2.375	N/A	0.875
$x_2 = 2$	1.5	0.375	N/A

For $I = 1$, we have $a_1 = 0.375$ and $a_2 = 0.875$, and Eqn. (4) becomes $0.375(2 - x_1)^2 + 0.875(2 - x_2)^2$. We need to decide the order of queries or the exact value of x_i so that Eqn. (4) is minimized. As $k = 2$, we can enumerate the value of x_i. The result is shown in Table 3. We see that 0.375 is the minimum, which means we should send q_1 first and then q_2. In this case, $a_1 < a_2$ and Eqn. (4) is minimized when q_1 is send before q_2. This indicates that the order to send the queries and the order of a_is should be the same in order to minimize Eqn. (4). We have the following greedy algorithm.

Greedy Query Order Optimization. Sort a_i's in Eqn. (4) in ascending order, that is, $a_{p_1} < a_{p_2} < \cdots < a_{p_k}$ where $\{p_1, \cdots, p_k\} = \{1, \cdots, k\}$. Then, $x_{p_i} = i \; \forall i = 1, 2, \cdots, k$, that is, sending query q_{p_i} at time iI, minimizes Eqn. (4).

The proof is simple. According to *rearrangement inequality*, Eqn. (4) is minimal when $(k - x_{p_1})^2 > (k - x_{p_2})^2 > \cdots > (k - x_{p_k})^2$ or $x_{p_1} < x_{p_2} < \cdots < x_{p_k}$. As we can send k queries, so every x_i is positive if we need to minimize Eqn. (4). Thus $x_{p_i} = i \; \forall i = 1, 2, \cdots, k$.

After all queries have been sent in the order specified by vector x_i, the algorithm sleeps for τ time units and another loop starts from the first query in the list. Each loop lasts for $kI + \tau$ time units.

Sometimes the the number of queries we can issue in each synchronization loop is low, then, some elements will not be synchronized in a loop. In the subsequent loops, the content freshness of the unsynchronized elements in previous loops will become significantly bigger because they have big values of t in $\frac{1}{2}\lambda t^2$. So these unsynchronized elements will finally get synchronized as this will lead to a great decrease in the content freshness of local relation L.

4.4 Query Frequency Optimization

We send a query every I time units and k queries require kI time units. If we do not send these queries, the content freshness of L will become $kI|L|$ if the content freshness of L increases as a linear function of the time. Our target is to ensure that the content freshness of L does not go beyond $\alpha kI|L|$ where α is a threshold, i.e., $E[F(L;kI)] < \alpha kI|L|$.

For fixed L, $E[F(L;kI)]$ is a function of $\boldsymbol{x} = (x_1, \cdots, x_k)$ and I. Let $f(\boldsymbol{x}, I) = E[F(L;kI)]$, $\tilde{\boldsymbol{x}} := \arg\min_{\boldsymbol{x}} f(\boldsymbol{x}, I)$, and $g(\boldsymbol{x}, I) = \frac{1}{kI|L|} f(\boldsymbol{x}, I)$, then the target becomes $g(\boldsymbol{x}, I) < \alpha$, where $g(\boldsymbol{x})$ is given as

$$g(\boldsymbol{x}, I) = \frac{1}{|L|^2} \sum_{i=1}^{k} \sum_{e_j \in q_i} \frac{w(e_j)\lambda_j (k - x_i)^2}{2k} I$$

$g(\boldsymbol{x}, I)$ is an increasing function of I for fixed $\boldsymbol{x}$ and L. If $I_1 > I_2 > 0$, we have

$$\begin{aligned} g(\tilde{\boldsymbol{x}}_1, I_1) &> g(\tilde{\boldsymbol{x}}_1, I_2) && (5) \\ &= \frac{1}{kI_2|L|} f(\tilde{\boldsymbol{x}}_1, I_2) && (6) \\ &\geq \frac{1}{kI_2|L|} f(\tilde{\boldsymbol{x}}_2, I_2) && (7) \\ &= g(\tilde{\boldsymbol{x}}_2, I_2) && (8) \end{aligned}$$

(5) is because g is an increasing function of I. (7) is based on the definition of $\tilde{\boldsymbol{x}}$. (6) and (8) are both based on the definition of g.

We start with a random value of I_0 (a simple choice is 1), and we calculate the correspondent $\tilde{\boldsymbol{x}}_0$ and the value of $g(\tilde{\boldsymbol{x}}_0, I_0)$. If the value is small than α, we use $2^1 I_0, 2^2 I_0, \cdots,$ $2^s I_0$ until $g(\tilde{\boldsymbol{x}}_s, 2^s I_0)$ is bigger than α; otherwise we use $2^{-1} I_0, 2^{-2} I_0, \cdots, 2^s I_0$. After the process, we have

$$g(\tilde{\boldsymbol{x}}_{s-1}, 2^{s-1} I_0) \leq \alpha \leq g(\tilde{\boldsymbol{x}}_s, 2^s I_0), s \in \mathbb{Z}$$

Then we just do a binary search in the interval $[2^{s-1} I_0, 2^s I_0]$ to get the final value of I.

5. EXPERIMENTS

In this section we describe the implementation of our prototype and present an experimental evaluation of our greedy synchronization method.

5.1 Settings

Methods. Our implementation considers all possible keyword subsets up to a fixed size. We found empirically that considering all subsets up to size two is already sufficient for good performance. A keyword query often returns multiple pages of ranked results. We use two methods to extract results from them.

Method 1 Scan through all result pages to extract as many tuples in the local relation as possible.

Method 2 Scan only the first page of results.

Data sets. We present experimental results for two application scenarios: paper citations and on-line DVD stores. For the paper citation scenario, we used the real citation data from Google scholar as the server database. To estimate the server coverage in the query efficiency heuristics, we used statistics extracted from DBLP. We also generated synthetic citation update data for the experiments measuring content freshness since it is not possible to obtain update and the data generation procedure will be described in Sec. 5.2. For the on-line DVD stores, we used real DVD pricing data from Amazon.com as our server database and the

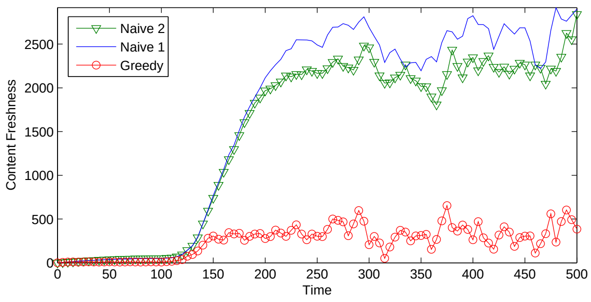


Figure 2: Content Freshness of Different Methods with the Same Query Frequency

statistics extracted from Internet Movie Database[3] (IMDB) to estimate the query efficiency heuristics.

Performance Measures. We use three different measures in our experiments. *Content freshness* measures the up-to-dateness of the local database with respect to a server database. Since update characteristics of the data elements on the server database cannot be observed, we use the formula with Poisson approximation in Eqn. (3). *Coverage ratio* measures the expected fraction of the number of tuples in the local relation covered by a set of queries Q,

$$\text{Coverage Ratio} = \frac{\sum_{q \in Q} E_{coverage}(q)}{|L|}.$$

$E_{coverage}(q)$ is the expected number of tuples in the local relation associated with query q. In Method 1, $E_{coverage}(q)$ is $|R_L(q)|$. In Method 2, $E_{coverage}(q)$ is computed as,

$$E_{coverage}(q) = \frac{p \cdot |R_L(q)|}{|R_S(q)|},$$

where p is the number of tuples in a result page and the server coverage term $|R_S(q)|$ is estimated using statistics (eg. $|R_S(q)| \propto |R_{DBLP}(q)|$ in our citation scenario). Finally, *Speedup ratio* measures the number of queries send to the server divided by the expected number of tuples in the local relation covered by the queries,

$$\text{Speedup Ratio} = \frac{\sum_{q \in Q} N(q)}{|\cup_{q \in Q} R_L(q)|}.$$

$N(q)$ is the total number of queries send to the server for query q. In Method 1, $N(q)$ is computed as

$$N(q) = \frac{|R_S(q)|}{p},$$

where p is the number of tuples in a result page. In Method 2, $N(q)$ is simply 1 for all $q \in Q$. *Coverage ratio* measures the number of tuples in the local relation covered by Q. Then for the tuples covered by Q, we need $|\cup_{q \in Q} R_L(q)|$ simple queries (i.e., ID based query) to synchronize them all. However, using our method, only $\sum_{q \in Q} N(q)$ keyword based queries are needed. *Speedup ratio* measures the speedup between these two methods. These two measures should be considered together as we need both high coverage ratio and high speedup ratio.

[3]Internet Movie Database: `http://www.imdb.com`

5.2 Content Freshness

In this experiment we investigate the performance in terms of content freshness for the "Naive 1", "Naive 2" and "Greedy" refresh algorithms. "Naive 1" sends simple queries (that is, ID based query). "Naive 2" also sends simple queries while the λ of every corresponding item is less than 1 (i.e., the item does not change too frequent). "Greedy" sends keyword queries picked by our approach.

The experiment is performed under the paper citations scenario. We generate synthetic citation data for this experiment based on statistics from the *Physical Review* journal [12]. The generated data contains 1000 papers with a citation distribution [12] following a power law, $NC(x) \sim x^{-\alpha}$, with $\alpha \approx 3$. We set the minimum number of citation to one, and use $\alpha = 3$ to get the normalize factor 2(i.e., calculate the constant factor in the citation distribution for $\int_{x=1}^{\infty} NC(x) = 1$). So our citation distribution is $NC(x) = \frac{2}{x^3}$. Every paper is assigned an initial citation number from the $NC(x)$ distribution. [13] uses attachment rate A_k to characterize the development of citations. A_k gives the likelihood that a paper with k citations will be cited by a new article. The data of 110 years of *Physical Review* suggests that A_k is a linear function of k(the scale factor is about 0.25), especially for k less than 150. We choose A_k to be $0.25k$, thus the λ of every paper is equal to a quarter of the initial citation number. As the update model of paper citation is assumed to be a Poisson process, we generate the citation update events of every paper based on its λ. In order to simulate the grouping effect of the GREEDYPROBES algorithm, we generate a group number for each paper. We have 200 groups while each can cover at most 10 papers. On average, a group covers about 5 papers. Then the papers in a group are synchronized by one query together.

Fig. 2 shows the content freshness of different refresh algorithms with the same query frequency. The "Naive 1" line, "Naive 2" line and "Greedy" line represent the content freshness of sending 10 queries using the corresponding algorithm. During the first 100 time units, the content freshness of these methods looks similar. From 100 time unit to 500 time unit, the content freshness of the "Naive 1" and "Naive 2" line experiences a stepped climbing procedure. Both line grow over 2000 at time 200 while the "Greedy" line is only about 200. Using our greedy synchronization strategy, more than 10 papers are synchronized during a time unit while the other two methods synchronize only 10 papers. So the content freshness of the "Greedy" line is much lower. Although the "Naive 1" method and "Naive 2" method synchronize the same number of papers during a time unit, but some papers will never get synchronized due to the synchronization strategy of the "Naive 2" method. When the time is long enough, the content freshness of these unsynchronized elements become extremely big and dominate the final value of the content freshness of the set. So the content freshness of "Naive 2" line finally goes beyond the "Naive 1" line. It keeps growing over 2500 at 500 time unit and appears to keep growing in the later time. The content freshness of "Greedy" line is controlled under 600 and the average content freshness is only about 350. It is clear that "Greedy" algorithm performs the best among these three methods.

5.3 Quality of Query Probes

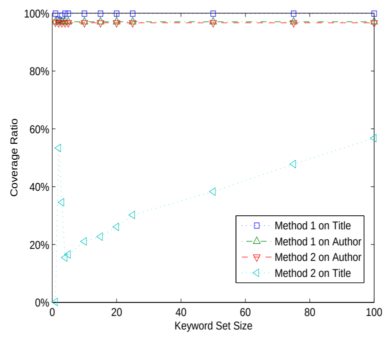
In the next series of experiments, we investigate the performance of our greedy synchronization approach by looking at the quality of the query probes generated by our GREEDYPROBES algorithm at each refresh event. The quality of a set of query probes is measured using the coverage ratio and speedup ratio.

Paper citations scenario. In the paper citations scenario, we want to maintain the citations of a set of papers. We use the DBLP data dump to estimate the value of $R_S(q)$ for query q. $R_{DBLP}(q)$ is the set of papers in the DBLP data associated with query q. Assume the ratio of $\frac{|R_S(q)|}{|R_{DBLP}(q)|}$ is a specific constant c. Based on the results of several queries, we estimate the constant c as 4.6.

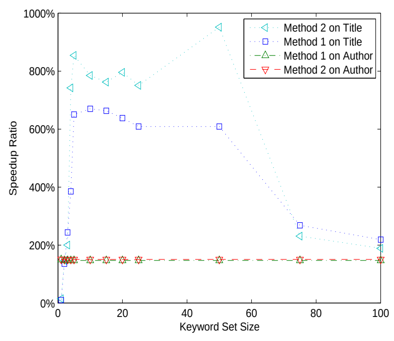
We create a paper set with 671 papers on 7 topics from the Google Scholar search result. Title and author are the attributes used in the experiment.

Using author as the keyword, we make a bag to hold all authors related to not covered papers. Then these authors are ranked based on their occurrences. The top-k frequent authors are selected to make up the keyword set $\mathcal{K}$. Smaller k makes the GREEDYPROBES algorithm faster while bigger k one is slower. But bigger k one is more likely to find an optimal solution as it tries more situation than smaller k one. So we change the value of k to see the value of the coverage ratio and the speedup ratio. Then we select the smallest k which achieves a high enough coverage and speedup ratio, i.e., we use the smallest computational resource to execute the GREEDYPROBES algorithm. The result of different k is shown in Fig. 3(a) and Fig. 3(b). The "Method 1 on Author" line and "Method 2 on Author" line in Fig. 3(a) are overlapping. Both line stay at the same value. So k does not affect the coverage ratio criteria in this case. As for the speedup ratio, the value of Method 1 remains the same while the value of Method 2 also remains the same. Method 2 performs slightly better than Method 1 on speedup ratio. To sum up, the keyword set size does not affect the performance of the algorithm for the author attribute. So directly choosing the most frequent author as a query will get a satisfactory result.

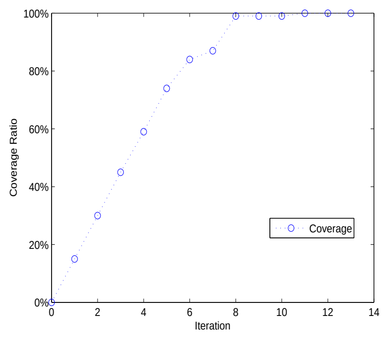
Experiment on title is carried out similarly. Every title is broken into single words and collected by a bag. Stopwords are removed from the bag and the remaining words are still ranked by their occurrences. The top-k frequent words are selected as the keyword set. Result is also shown in Fig. 3(a) and Fig. 3(b). When the keyword size is 1 ($k = 1$), the algorithm will always return single word queries as the keyword set has only one word. But the selectivity of single word queries from titles is extremely low, that is, you will get millions of papers if you send a single word query from titles to the search engine. It is easy to see that single word queries from titles can always cover the local paper set. So the coverage ratio of "Method 1 on Title" is 100%. But if you use Method 2 to retrieve the results, you will get an extremely low coverage ratio as the expected number of papers in the first result page is small. In the experiment, the coverage ratio of "Method 2 on Title" is nearly zero when k is 1. However, when the keyword size is exactly 2 ($k = 2$), the iteration procedure is more likely to return query of two keywords as the selectivity of two keywords query is much higher than single word query. The "Method 2 on Title" line starts from about 55%, drops below 20% immediately, and then increases to about 60% as k varies from 2 to 100. Intuitively, large keyword set size is more likely to return a



(a) Coverage Ratio

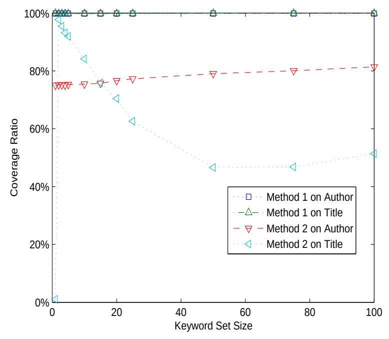


(b) Speedup Ratio

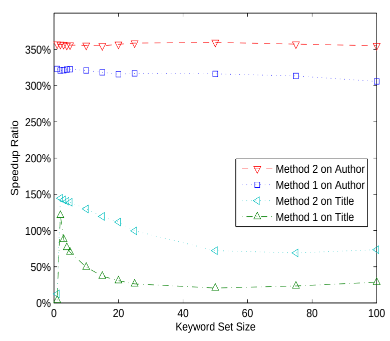


(c) Coverage Ratio Growth (Top 5 frequent keyword from titles & Method 1)

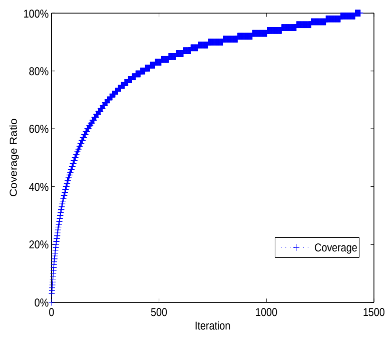
Figure 3: Coverage ratio, speedup ratio and coverage ratio growth of the Google paper citations data set. Note that in the coverage ratio plot, the lines for "Method One on Author" and "Method Two on Author" are overlapping, and in the speedup plot, the lines for Method One on Author" and "Method Two on Author" are overlapping.



(a) Coverage Ratio



(b) Speedup Ratio



(c) Coverage Ratio Growth (Top 1 frequent keyword from authors & Method 1)

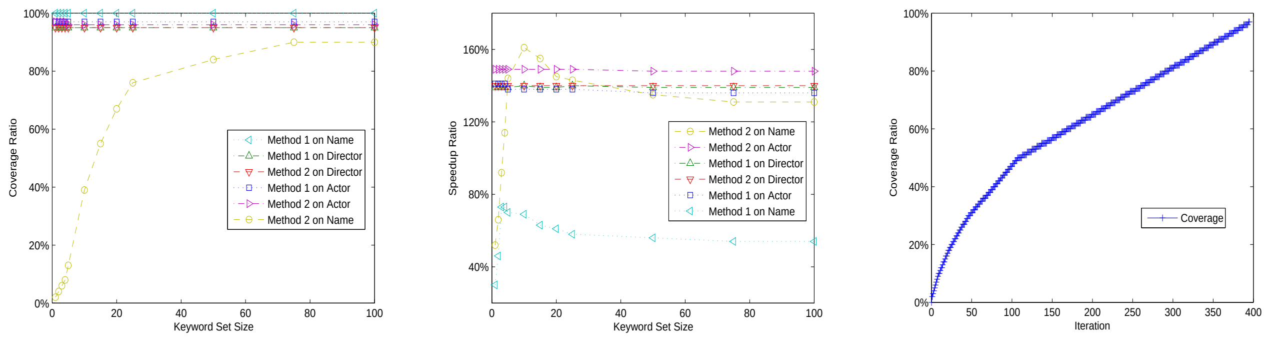
Figure 4: Coverage ratio, speedup ratio, and coverage ratio growth of Microsoft Research paper citations data set. Note that in the coverage ratio plot, the lines for "Method One on Author" and "Method One on Title" are overlapping.

better group of queries in the query generation. But in this case, the performance of keyword size set 75 is worse than that of keyword size set 2 and the performance of keyword size set 100 is roughly the same as that of keyword size set 2. The reason lies in the query generation algorithm is a greedy one. Greedy algorithm does not guarantee to return an optimal solution as there is chance for the selection procedure to return a query which does not lead to the optimal solution.

Similar for the coverage ratio, the speedup ratio of Method 1 and Method 2 on title is nearly zero when the keyword size is 1. This is also because of the extremely low selectivity of single keyword query. The speedup ratio of Method 1 and Method 2 on title changes a lot in Fig. 3(b). The reason also lies in the algorithm is a greedy one. As the coverage ratio line of "Method 1 on Title" stays at about 100% for all keyword set size, then the keyword set size which achieves the highest speedup ratio performs best for "Method 1 on title". In this case, the speedup ratio of keyword size set 5 and keyword size set 10 are roughly the same. So we choose the smaller one 5 to achieve a better time efficiency.

As the coverage ratio of "Method 2 on Title" is low, the speedup ratio of "Method 1 on Author" and "Method 2 on Author" are lower than that of "Method 1 on Title", Method 1 on title with top 5 frequent keyword (keyword set size 5) achieves the best performance in this case. The growth of coverage ratio as the iteration time increases in this configuration is shown in Fig. 3(c). During the first 8 iteration, high coverage queries are generated, so the curve increases sharply. The last five iterations generate relatively lower coverage queries, so the curve increases smoothly.

We also conduct experiments on larger data sets. 6839 papers from Microsoft Research are used as the local data set. In this case, affiliation is the best attribute to make a query, i.e., you can simply send "Microsoft Research" to the search engine to get the list of all these papers. However, Google Scholar does not support affiliation as the search attribute. So we still have to use title and author as attributes to make queries. Result is shown in Fig. 4(a) and Fig. 4(b). The best performance is taken when using top-1 frequent keyword from authors and the coverage ratio growth of this best performance case is shown in Fig. 4(c). In this case, the performance of author attribute is still very stable while the performance of title attribute changes a lot. When the keyword set size is 1, the title attribute achieves the worst performance for the low selectivity of single keyword query. However, there is still difference between the performance of the two data sets. In the Google paper citations data set, the best performance is taken on the title attribute. But in the Microsoft Research paper citations data set, the best



(a) Coverage Ratio

(b) Speedup Ratio

(c) Coverage Ratio Growth (Top 1 frequent keyword from actors & Method 1)

Figure 5: Coverage ratio, speedup ratio and coverage ratio growth for the Amazon price data set. Note that in the coverage ratio plot, the lines for "Method One on Director", "Method Two on Director" "Method One on Actor" and "Method Two on Actor" are overlapping, and in the speedup plot, the lines for "Method One on Director", "Method Two on Director" and "Method One on Actor" are overlapping.

performance is taken on the author attribute. The difference between the results reveals the characters of the data set. As the Google paper citations data set is generated using several topics, there is a lot of common words in the titles of the papers. So the performances of title attribute is the best. The Microsoft Research paper citations data set is a collection of papers published by researchers in Microsoft and many papers share the same author in the data set. So the performance of author attribute is the best.

The Online DVD Stores Scenario. In the online DVD stores scenario, we keep track of the price information of a movie set through synchronization with the Amazon.com data source. We use the IMDB (an online database of information related to movies, actors, television shows, etc.) statistics to estimate the value of $R_S(q)$ as $0.58 \cdot |R_{IMDB}(q)|$. Name, director and actor are attributes chosen to carry out the experiment. Results on these three attributes are shown in Fig. 5(a) and Fig. 5(b). The best performance is achieved when using top 1 frequent keyword from actors and the coverage ratio growth is shown in Fig. 5(c).

6. CONCLUSION

Many applications keep local copies of data that is frequently updated on the web. In this paper, we introduce the concept of content freshness as opposed to the traditional concept of data freshness. Our goal is to keep a local copy of remote data up-to-date. Unlike previous approaches, we take advantage of the semantics of the data in designing a synchronization strategy which gives priority to highly important and/or volatile data. Assuming the remote data is only accessible via keyword search, we solve an optimization problem that minimizes the number of keyword searches while maximizing the up-to-dateness of the local data.

7. REFERENCES

[1] L. Barbosa and J. Freire. Searching for hidden-web databases. In *Proceedings of WebDB*, volume 5, pages 1–6. Citeseer, 2005.

[2] L. Barbosa and J. Freire. An adaptive crawler for locating hidden-Web entry points. In *World Wide Web*, pages 441–450. ACM New York, NY, USA, 2007.

[3] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. *SIGMOD Rec.*, 29(2):117–128, 2000.

[4] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, pages 265–274. ACM New York, NY, USA, 2001.

[5] Ping Li and Kenneth W. Church. A sketch algorithm for estimating two-way and multi-way associations. *Comput. Linguist.*, 33(3):305–354, 2007.

[6] Hao Liang, Wanli Zuo, Fei Ren, and Junhua Wang. Translating query for deep web using ontology. In *CSSE (4)*, pages 427–430, 2008.

[7] Jianguo Lu, Yan Wang, Jie Liang, Jessica Chen, and Jiming Liu. An approach to deep web crawling by sampling. In *Web Intelligence*, pages 718–724, 2008.

[8] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Y. Halevy. Google's deep web crawl. *PVLDB*, 1(2):1241–1252, 2008.

[9] A. Ntoulas, P. Pzerfos, and J. Cho. Downloading textual hidden web content through keyword queries. In *Digital Libraries, 2005. JCDL'05. Proceedings of the 5th ACM/IEEE-CS Joint Conference on*, pages 100–109, 2005.

[10] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, pages 73–84. ACM New York, NY, USA, 2002.

[11] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB*, pages 129–138, 2001.

[12] S. Redner. How popular is your paper? An empirical study of the citation distribution. *The European Physical Journal B*, 4(2):131–134, 1998.

[13] S. Redner. Citation statistics from 110 years of Physical Review. *Physics today*, 58(6):49–54, 2005.

[14] K.C. Sia, J. Cho, and H. Cho. Efficient monitoring algorithm for fast news alerts. *TKDE*, 19(7):950, 2007.

[15] V.V. Vazirani. *Approximation algorithms*. Springer, 2004.

[16] Yan Wang, Jianguo Lu, and Jessica Chen. Crawling deep web using a new set covering algorithm. In *ADMA*, pages 326–337, 2009.

[17] P. Wu, J.R. Wen, H. Liu, and W.Y. Ma. Query selection techniques for efficient crawling of structured web sources. In *ICDE*, pages 47–47, 2006.