# Storing the Evolving Web Efficiently

Xixuan Feng
University of Wisconsin-Madison
Madison, WI 53706, USA
xfeng@cs.wisc.edu

## ABSTRACT

Many web applications operate on large amount of web data. How to store and access the data itself is becoming a serious problem, especially with the dynamic of the web. This paper introduces an approach that helps applications to manage evolving web data efficiently, saving a lot of storage space with fast access to the data. One important feature of this framework is that the worst case of random access time is bounded by a simple parameter, and this enable application specific systems based on it. The results on 3 syntactic datasets and 2 real-world datasets show that the approach adaptively performs well.

## 1. INTRODUCTION

The web has achieved great development since it was invented over two decades ago. The question, about how to manage the large amount of data on the web, is always drawing much attention, in both industry and academia. Techniques like information retrieval, information extraction and knowledge discovery have been very popular in the data management community recently. Efficiently access to web data is essential for all of above methods.

While some traditional applications such as search engine mainly make use of the current snapshot of the web, historical web data is necessary in terms of time-dependent use of the web (e.g., news trends). In addition, even if a web application can run with only the newest snapshot of web data, historical data is important in the aspects of debugging, undoing and evaluation. However, due to the fast development of the web, storing historical web data can be extremely costly. For example, DBLife [2] operates on over 10,000 URLs and more than 120MB of data is crawled every day [1]. This university application with less than 10 maintainers can generate up to 50GB of data for one year, without even considering the significant growth of the web.

In this paper, we purpose a framework that not only removes duplication between web data snapshots in order to save storage space, but also provides fast access to both current and past snapshots. An upper bound of access time will be proved.

As is shown in Figure 1, p1 and p2 are webpage snapshots of the same URL (the place where some database group or researcher lists its publications), but crawled at different time. When storing p2 in the presence of p1, our method reuses c1 and c2 by setting up p1 as the reference of p2 and saves only the difference (also noted as delta) c3.

```
 ____________________________
|Publications                |--c1
|    SIGMOD-09               |--.
|    SIGMOD-09               |  c2
|    SIGMOD-09               |--'
|    ...                     |
|                            |
|____________________________|
          p1
 ____________________________
|Publications                |--c1
|    ICDE-10                 |--c3
|    SIGMOD-09               |--.
|    SIGMOD-09               |  c2
|    SIGMOD-09               |--'
|    ...                     |
|____________________________|
          p2
```

**Figure 1: Two pages crawled at different time.**

In the remaining sections of this paper, some of the challenges and how the framework handles each of them is discussed in section 2. Section 3 gives a formal definition of the problem our framework solves. And the solution follows in section 4. The evaluation is presented in section 5. In section 6, there is a summary that what I have learned in the project and how the approach may be improved.

## 2. CHALLENGES

The idea seems to be simple, but there are several difficulties in designing an optimal solution. First, storing delta of a reference instead of full text costs extra time to recover the original file if the operation processes the data randomly. We introduce a parameter $k$, guaranteeing that no pages rely on more than $(k-1)$ other pages to be constructed. This allows applications to choose the trade-off between storage space and random access time. The system with a larger $k$ saves more space but requires longer average random access time.

Second, it's clear that it is a common case that many consecutive snapshots share the same URL also have a lot of overlapping content. However, the case is not guaranteed.

Storing the difference between 2 completely distinct pages consumes more storage space than either full-text of the two pages. Our method includes a detector to identify such a case. When this happens, we store full-text instead of the difference between the two.

## 3. PROBLEM DEFINITION

Let $U = u_1, u_2, ..., u_m$ be a set of URL operated by an information management system $M$. For instance, DBLife considers a set of URL that relates to the database community. We consider the case that $M$ crawls these URLs at regular time interval and generates $V_i$, the set of webpage snapshots, at time interval $i$.

Our problem definition can be stated as follows. Given any $u$ in $U$, let $p_1, p_2, ..., p_n$ denote the consecutive snapshots (retrieved at regular time interval) of the webpage with the URL $u$, and $p_{n+1}$ be the snapshot next to $p_n$. Therefore, the problem come down to minimize the storage space used to save $p_{n+1}$ into the database, and at the same time, provide a way to load $p_1, ..., p_{n+1}$ into memory fast enough.

## 4. SOLUTION

First, we design the following 3 tables to save the webpages: urls (uid serial, url text UNIQUE), snapshots (pid serial, uid integer NOT NULL, timestamp date, UNIQUE (uid, timestamp)), content (pid integer NOT NULL UNIQUE, delta text, ref_id integer). Table urls is storing URLs with IDs. Table snapshots identifies a single page file with its URL and timestamp. Finally, Table content is the table which the file data actually locates. Deltas are of form (place_to_insert, text). Besides delta, it also specifies which snapshot should the delta be applied to (reference snapshot). It is important to notice that the ref_id can be null, which indicates the full-text is stored. The following figure illustrates the layout of Table content with the data in the example in Figure 1.

```
 ____________________________________________
| p1 | (0, "Publication\n...") | NULL |
|----|-------------------------|------|
| p2 | (12, "ICDE-10\n")       |  p1  |
|----|-------------------------|------|
|                 ...                 |
|_____________________________________|
            Table content
```

**Figure 2: Table content layout.**

Next, in order to provide fast recovery of snapshots, we group no more than $k$ ($k$ is a parameter chosen by users) consecutive snapshots with the same URL into one cluster, and the first snapshot of a cluster is stored in the database as full-text. This provides an upper bound of random access to any snapshots, because, in the worst case, at most 1 full-text and k deltas are loaded and k times of recovery are performed. In addition, the number of snapshots in one cluster can be less than $k$ when two snapshots generate a delta that larger than the newest snapshot, in which case we start a new cluster.

Thirdly, it is about how we choose the reference page for each page. Intuitively, picking the previous adjacent snapshot is a simple and effective way, given that webpages are devised little by little. However, we consider another common case: modify-undo. People often use a fixed template to represent the idle state. For example, notification boards contain only page frame when no notifications are available. If we always use the previous snapshot as reference snapshot, multiple copies of this sort of templates will be stored. Therefore, we choose a snapshot in the same cluster but giving a minimum delta as the reference snapshot. Although we relax the constraint by choosing a reference snapshot other than the adjacent one, the upper bound of random access time still holds.

## 5. EVALUATION

In this section, we compare three slightly different implementations of the framework: a) first-reference, which always chooses the first version in a cluster to be the reference page; b) previous-reference, which always chooses the consecutive previous version to be the reference page; c) best-reference, which always chooses the version the with the smallest delta to be the reference page. We first show the results on 3 different syntactic data sets trying to analyze what kinds of data each implementation performs well on. And then we evaluate them using 2 real-world data sets, to see how useful they are. A postgresql database server is used to set up relational tables and store data.

### 5.1 Syntactic Datasets

#### *5.1.1 Stable Pages*

Even though the web is dynamic, when we deal with large amount of data, it is very likely that some pages never change after some time. The first syntactic data set contains 1,000 identical pages, each of size 7.1KB. Figure 3

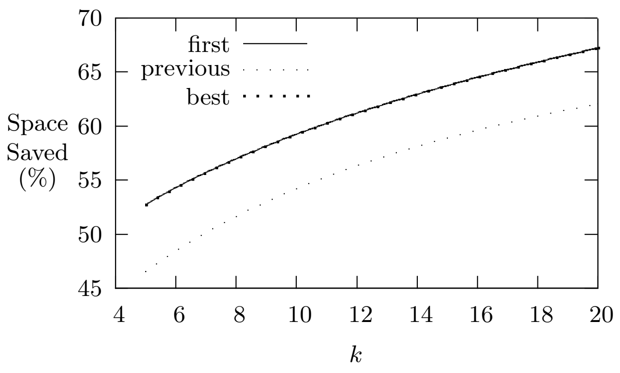

**Figure 3: Stable Pages.**

shows what percentage of storage space has been saved using 3 implementations upon k. The curves of first-reference and best-reference are exactly the same because they always choose the same references. More ref_id are needed to be stored, so previous-reference does not perform as good as the other two. Even with a small $k$, about 50% of the storage space can be saved in any implementations. It's worth noticing that postgresql has built-in compression, and all of our results show improvement upon compressed data. This means we have captured some aspects of saving space, which the compression algorithm is not able to capture.

### 5.1.2 *Notification Boards*

The second type of syntactic data contains pages that all have a same template, with one version only template but no content, the next one filled with random content, and then empty content with template only again, etc. It simulates some web rss applications, e.g. Google Reader. Each page has the template of size 5.0KB, and possibly content of size ranged from 0.1 to 10KB. From Figure 4, previous-

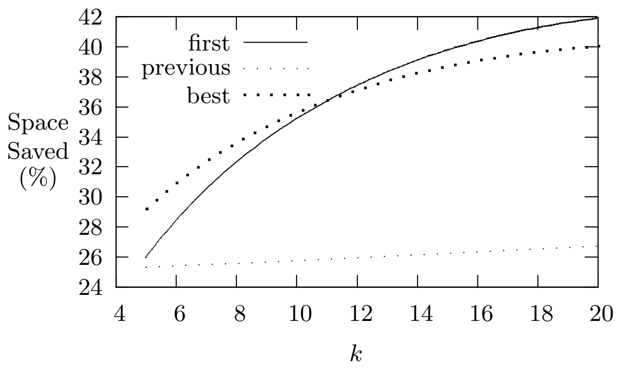

**Figure 4: Notification Boards.**

reference is not good at such modify-undo data set. As to the best-reference, it saves more space than first-reference with a small *k*. However, it saves less than first-reference when *k* is large enough. This would not happen without compression, because the gap comes from smaller cardinality of the ref_id of first-reference.

### 5.1.3 *Incrementally Adding*

Commenting boards and forum topics with many replies are both very common on the web. In this data set, each page has a random string (including empty string) in addition to the consecutive previous one. This data set fits previous-

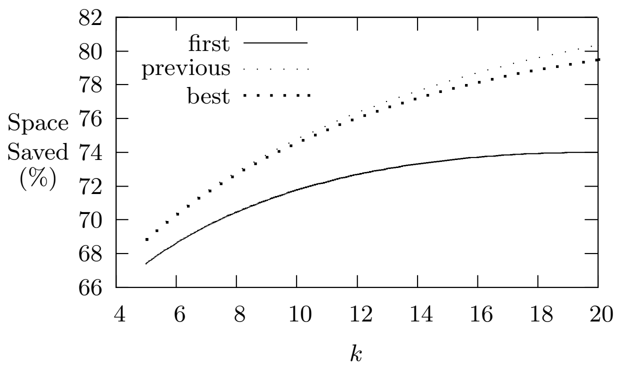

**Figure 5: Incrementally Adding.**

reference perfectly.

## 5.2 Real-World Data

### 5.2.1 *Website Homepages*

I crawled 4 homepages from main portal and news sites, including Yahoo, AOL, CNN and NYTimes. Because the time constraint, the time interval I picked is one hour. That's also the reason why I choose such dynamic homepages. The total size of files is about 140MB, and it costs 32MB in database after compression. Figure 6 indicates that the best-reference

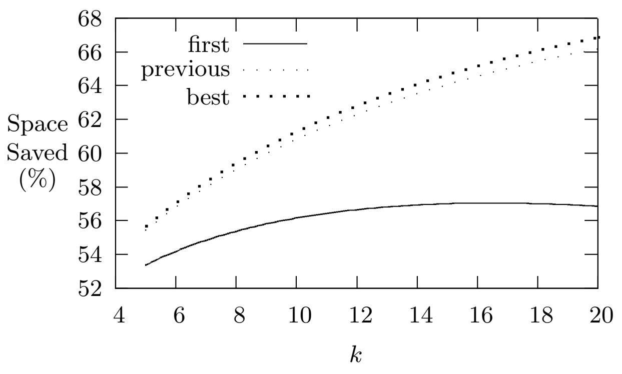

**Figure 6: Homepages.**

is more adaptive in real-world data, but it takes more time to choose the smallest delta. When k becomes larger, this can be worse, but, of course, it can be done totally offline after all.

### 5.2.2 *Real Incrementally Adding*

The last data set is crawled from digg.com, a news integration web sites with many postings and replies. It should verify the result of the Incrementally Adding syntactic data. Even the data is in the form of incrementally adding, best-

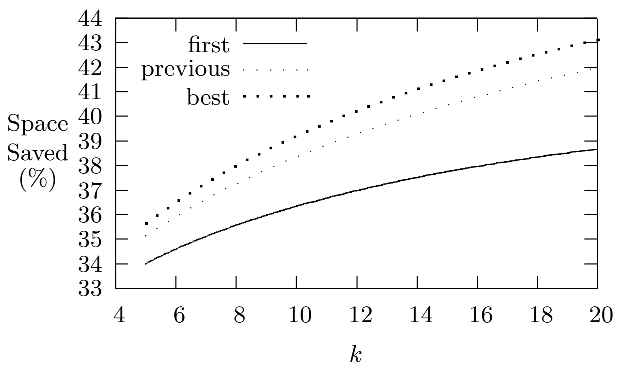

**Figure 7: Real Incrementally Adding.**

reference performs better at this one. It shows the importance of adaptability in the real world.

## 6. SUMMARY

The evolving web is valuable. To use it, the first step is to store it. We have presented a framework to improve the way to store these evolving data. Our evaluation shows that this framework takes good advantage of the evolving data, and it saves a significant portion of storage space even upon the compressed data.

There are a lot to improve this work. 1) Instead of bothering the applications or users, how to set $k$ as using the knowledge from the data? 2) Use a better matcher to get smaller delta. 3) More work on measuring random access time. 4) Integrate the optimization of storing stable pages. 5) etc.

## 7. REFERENCES

[1] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. In *ICDE*, pages 943–952. IEEE, 2008.

[2] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community (demo). In *CIDR*, pages 169–172. www.crdrdb.org, 2007.