Synchronizing a database to Improve Freshness

Junghoo Cho Hector Garcia-Molina
Stanford University
{cho, hector}@cs.stanford.edu

Abstract

In this paper we study how to refresh a local copy of an autonomous data source to maintain the copy up-to-date. As the size of the data grows, it becomes more difficult to maintain the copy "fresh," making it crucial to synchronize the copy effectively. We define two freshness metrics, change models of the underlying data, and synchronization policies. We analytically study how effective the various policies are. We also experimentally verify our analysis, based on data collected from 270 web sites for more than 4 months, and we show that our new policy improves the "freshness" very significantly compared to current policies in use.

1 Introduction

Local copies of remote data sources are frequently made to improve performance or availability. For instance, a data warehouse may copy remote sales and customer tables for local analysis. Similarly, a web search engine copies portions of the web, and then indexes them to help users navigate the web. In many cases, the remote source is updated independently without pushing updates to the client that has a copy, so the client must periodically poll the source to detect changes and refresh its copy. This scenario is illustrated in Figure 1.

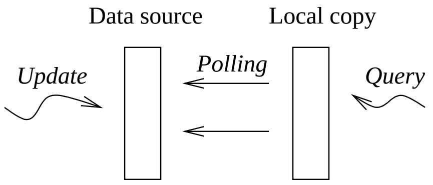


Figure 1: Conceptual diagram of the problem

Clearly, a portion of the local copy may get temporarily out-of-date, due to the delay between source updates and the refresh of the local copy. In many applications it may be important to control how out-of-date information becomes, and to perform the refresh process so that data "freshness" is improved. In this paper we address some important questions regarding this refresh or synchronization process. For instance, 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? (Surprisingly, the answer to this last question is *no* in some cases!)

Although the synchronization and freshness problem arises in various contexts, our work is driven by the need to manage web data. At Stanford, we maintain a local repository called WebBase, containing a significant portion of the web (currently 42 million pages), that supports researchers experimenting with web searching and mining [10, 5]. (The Google search engine used this repository before it became a commercial product [1].) Web search engines and services, such as Alexa, AltaVista and Infoseek, also maintain similar copies of the web, or indexes based on the web data collected. To maintain the repository and/or index up-to-date, the web pages must be periodically revisited. This work is done by a program called a *web crawler*.

As the size of the web grows rapidly, it becomes crucial to synchronize the data more effectively. A recent study shows that it takes up to 6 months for a new page to be indexed by popular web search engines [9]. Also, a lot of users express frustration, when a search engine returns obsolete links, and the users follow the links in vain. According to the same study, up to 14% of the links in the search engines are broken. By tuning the synchronization policy, we believe we can reduce the wasted resources and time significantly.

The effective synchronization of a local copy introduces many interesting challenges. First of all, measuring the freshness of the copy is not trivial. Intuitively, the copy is considered fresh when it is not different from the "real-world" remote data. Therefore, we can measure its freshness only when we know the *current status* of the real-world data. But how can we know the current status of the real-world data, when it is spread across thousands of web sites? Second, we do not know exactly when a particular data item will change, even if it changes at a certain average rate. For instance, the pages in the CNN web site are updated about once a day, but the update of a particular page depends on how the news related to that page develops over time. Therefore, visiting the page once a day does not guarantee its freshness.

In this paper, we will formally study how to synchronize the data to maximize its freshness. The main contributions we make are:

- We present a formal framework to study the synchronization problem, and we define the notions of freshness and age of a copy. While our study focuses on the web environment, we believe our analysis can be applied to other contexts, such as a *data warehouse*. In a warehouse, *materialized views* are maintained on top of *autonomous* databases, and again, we need to *poll* the underlying database periodically to guarantee some level of freshness.
- We present several synchronization policies that are currently employed, and we compare how effective they are. Our study will show that some policies that may be intuitively appealing might actually perform *worse* than a naive policy.
- We also propose a new synchronization policy which may improve the freshness by orders of magnitude in certain cases.
- We validate our analysis using experimental data collected from 270 web sites over 4 months. The data will show that our new policy is indeed better than any of the current policies.

The rest of this paper is organized as follows. In Section 2, we present a framework for the synchronization problem. Then in Section 3, we explain what options exist for synchronizing the local copy, and we compare these options in Section 4 and 5. In Section 6, we verify our analysis using data collected from the world wide web.

2 Framework

To study the synchronization problem, we first need to understand the meaning of "freshness," and we need to know how the data change over time. In this section we present our framework to address these issues. In our discussion, we refer to the data source that we monitor as the *real-world database* and its local copy as the *local database* when we need to distinguish them. Similarly, we refer to their data items as the *real-world elements* and as the *local elements*.

In Section 2.1, we start our discussion with the definition of two freshness metrics, *freshness* and *age*. Then in Section 2.2, we discuss how we model the evolution of individual real-world elements. Finally in Section 2.3 we discuss how we model the real-world database as a whole.

2.1 Freshness and age

Intuitively, we consider a database "fresher" when the database has more up-to-date elements. For instance, when database A has 10 up-to-date elements out of 20 elements, and when database B has 15 up-to-date elements, we consider B to be fresher than A. Also, we have a notion of "age:" Even if all elements are obsolete, we consider database A "more current" than B, if A was synchronized 1 day ago, and B was synchronized 1 year ago. Based on this intuitive notion, we define *freshness* and *age* as follows:

1. **Freshness:** Let $S = \{e_1, \ldots, e_N\}$ be the local database with N elements. Ideally, all N elements will be maintained up-to-date, but in practice, only $M(< N)$ elements will be up-to-date at a specific time. (By up-to-date we mean that their values equal those of their real-world counterparts.) We define the *freshness* of S at time t as $F(S;t) = M/N$. Clearly, the *freshness* is the fraction of the local database that is up-to-date. For instance, $F(S;t)$ will be one if all local elements are up-to-date, and $F(S;t)$ will be zero if all local elements are out-of-date. For mathematical convenience, we reformulate the above definition as follows:

   Definition 1 The *freshness* of a local element e_i at time t is
   $$F(e_i;t) = \begin{cases} 1 & \text{if } e_i \text{ is up-to-date at time } t \\ 0 & \text{otherwise.} \end{cases}$$
   Then, the *freshness* of the local database S at time t is
   $$F(S;t) = \frac{1}{N}\sum_{i=1}^{N} F(e_i;t).$$
   □

   Note that freshness is hard to measure exactly in practice, since we need to "instantaneously" compare the real-world data to the local copy. But as we will see, it is possible to estimate freshness (and age) given some information about how the real-world data changes.

2. **Age:** To capture "how old" the database is, we define the metric *age* as follows:

   Definition 2 The *age* of the local element e_i at time t is
   $$A(e_i;t) = \begin{cases} 0 & \text{if } e_i \text{ is up-to-date at time } t \\ (t - \text{modification time of } e_i) & \text{otherwise.} \end{cases}$$
   Then the *age* of the local database S is
   $$A(S;t) = \frac{1}{N}\sum_{i=1}^{N} A(e_i;t).$$
   □

   The *age* of S tells us the average "age" of the local database. For instance, if all real-world elements changed one day ago and we have not synchronized them since, $A(S;t)$ is one day.

In Figure 2, we show the evolution of $F(e_i;t)$ and $A(e_i;t)$ of an element e_i. In this graph, the horizontal axis represents time, and the vertical axis shows the value of $F(e_i;t)$ and $A(e_i;t)$. We assume that the real-world element changes at the dotted lines and the local element is synchronized at the dashed lines. The *freshness* drops to zero when the real-world element

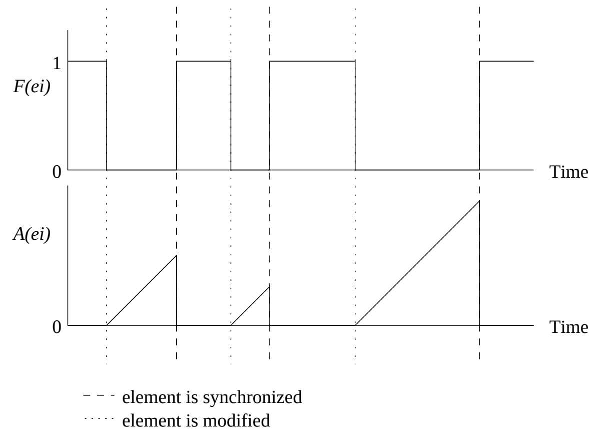


Figure 2: An example of the time evolution of $F(e_i;t)$ and $A(e_i;t)$

changes, and the *age* increases linearly from that point on. When the local element is synchronized to the real-world element, its *freshness* recovers to one, and its *age* drops to zero.

Obviously, the freshness (and age) of the local database may change over time. For instance, the freshness might be 0.3 at one point of time, and it might be 0.6 at another point of time. To compare different synchronization methods, it is important to have a metric that fairly considers freshness over a period of time, not just at one instant. In this paper, we use the freshness *averaged over time* as this metric.

Definition 3 We define the freshness of element e_i *averaged over time*, $\bar{F}(e_i)$, and the freshness of database S *averaged over time*, $\bar{F}(S)$, as

$$\bar{F}(e_i) = \lim_{t\to\infty} \frac{1}{t} \int_0^t F(e_i;t)dt$$
$$\bar{F}(S) = \lim_{t\to\infty} \frac{1}{t} \int_0^t F(S;t)dt.$$

The time average of age can be defined similarly. □

From the definition, we can prove that $\bar{F}(S)$ is the sum of $\bar{F}(e_i)$: $\bar{F}(S) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(e_i)$. For detailed proof, see [3].

2.2 Poisson process and probabilistic evolution of an element

To study how effective different synchronization methods are, we need to know how the real-world element changes. In this paper, we assume that the elements are modified by a *Poisson process*. A Poisson process is often used to model a sequence of events that happen *randomly* and *independently* with a *fixed rate* over time. For instance, the occurrences of fatal auto accidents, or the arrivals of customers at a service center, are usually modeled by *Poisson processes*. Under a Poisson process, it is well-known that the time to the next event is exponentially distributed [11].

Lemma 1 *Let T be the time when the next event occurs in a Poisson process with change rate λ. Then the probability density function for T is*

$$f_T(t) = \begin{cases} \lambda e^{-\lambda t} & \text{for } t > 0 \\ 0 & \text{for } t \leq 0. \end{cases}$$

□

In this paper, we assume that each element e_i is modified by the Poisson process with change rate λ_i. That is, each element changes at its own rate λ_i, and this rate may differ from element to element. For example, one element may change once a day, and another element may change once a year. Later in Section 6, we will experimentally verify that the Poisson process describes well the changes of *real* web pages.

Under the Poisson process model, we can analyze the freshness and age of the element e_i over time. More precisely, let us compute the *expected value* of *freshness* and *age* of e_i at time t. For the analysis, we assume that we synchronize e_i at $t = 0$ and at $t = I$.

By integrating the probability density function of Lemma 1, we can obtain the probability that e_i changes in the interval $(0, t]$:

$$\Pr\{T \leq t\} = \int_0^t f_T(t)dt = 1 - e^{-\lambda t}$$

Since e_i is not synchronized in the interval $(0, I)$, the local element e_i may get out-of-date with probability $\Pr\{T \leq t\} = 1 - e^{-\lambda t}$ at time $t \in (0, I)$. Hence, the *expected freshness* is

$$\mathrm{E}[F(e_i;t)] = 0 \cdot (1 - e^{-\lambda t}) + 1 \cdot e^{-\lambda t} = e^{-\lambda t} \quad \text{for } t \in (0, I).$$

Note that the expected freshness is 1 at time $t = 0$ and that the expected freshness approaches 0 as time passes.

We can obtain the *expected value* of *age* of e_i similarly. If e_i is modified at time $s \in (0, I)$, the age of e_i at time $t \in (s, I)$ is $(t - s)$. From Lemma 1, e_i changes at time s with probability $\lambda e^{-\lambda s}$, so the expected age at time $t \in (0, I)$ is

$$\mathrm{E}[A(e_i;t)] = \int_0^t (t-s)(\lambda e^{-\lambda s})ds = t\left(1 - \frac{1 - e^{-\lambda t}}{\lambda t}\right)$$

Note that $\mathrm{E}[A(e_i;t)] \to 0$ as $t \to 0$ and that $\mathrm{E}[A(e_i;t)] \approx t$ as $t \to \infty$; the expected age is 0 at time 0 and the expected age is approximately the same as the elapsed time when t is large. In Figure 3, we show the graphs of $\mathrm{E}[F(e_i;t)]$ and $\mathrm{E}[A(e_i;t)]$. Note that when we resynchronize e_i at $t = I$, $\mathrm{E}[F(e_i;t)]$ recovers to one and $\mathrm{E}[A(e_i;t)]$ goes to zero.

2.3 Evolution model of database

In the previous subsection we modeled the evolution of an element. Now we discuss how we model the database as a whole. Depending on how its elements change over time, we can model the real-world database by one of the following:

- **Uniform change-frequency model:** In this model, we assume that all real-world elements

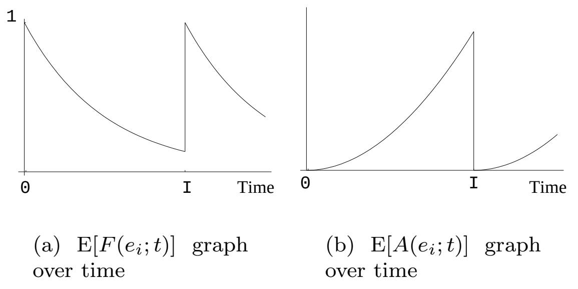


Figure 3: Time evolution of E[$F(e_i;t)$] and E[$A(e_i;t)$]

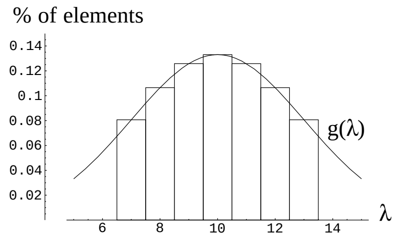


Figure 4: Histogram of the change frequencies

change at the *same* frequency λ. This is a simple model that could be useful when:

– we do not know how often the *individual* element changes over time. We only know how often the entire database changes *on average*, so we may assume that all elements change at the same *average* rate λ.

– the elements change at *slightly* different frequencies. In this case, this model will work as a good approximation.

- **Non-uniform change-frequency model:** In this model, we assume that the elements change at *different* rates. We use λ_i to refer to the the change frequency of the element e_i. When the λ_i's vary, we can plot the histogram of λ_i's as we show in Figure 4. In the figure, the horizontal axis shows the range of change frequencies (e.g., $9.5 < \lambda_i \leq 10.5$) and the vertical axis shows the fraction of elements that change at the given frequency range. We can approximate the discrete histogram by a continuous distribution function $g(\lambda)$, when the database consists of many elements. We will adopt the continuous distribution model whenever convenient.

For the reader's convenience, we summarize our notation in Table 1. As we continue our discussion, we will explain some of the symbols that have not been introduced yet.

3 Synchronization policy

So far we discussed how the real-world database changes over time. In this section we study how the local copy can be refreshed. There are several dimensions to this synchronization process:

1. **Synchronization frequency:** We first need to decide *how frequently* we synchronize the local database. Obviously, as we synchronize the database more often, we can maintain the local database fresher. In our analysis, we assume that we synchronize N elements per I time-units. By varying the value of I, we can adjust how often we synchronize the database.

2. **Resource allocation:** Even after we decide how many elements we synchronize per unit interval, we still need to decide how frequently we synchronize *each individual* element. We illustrate this issue by an example.

   Example 1 The database consists of three elements, e_1, e_2 and e_3. It is known that the elements change at the rates $\lambda_1 = 4$, $\lambda_2 = 3$, and $\lambda_3 = 2$ (times/day). We have decided to synchronize the database at the *total* rate of 9 elements/day. In deciding how frequently we synchronize each element, we consider the following options:

   - Synchronize all elements uniformly at the same rate. That is, synchronize e_1, e_2 and e_3 at the same rate of 3 (times/day).
   - Synchronize an element proportionally more often when it changes more often. In other words, synchronize the elements at the rates of $f_1 = 4$, $f_2 = 3$, $f_3 = 2$ (times/day). □

   Based on how the fixed synchronization-resource is allocated to the individual elements, we can classify synchronization policies as follows. We study these policies later in Section 5.

   (a) **Uniform allocation policy:** We synchronize all elements at the same rate, regardless of how often they change. That is, each element e_i is synchronized at the fixed frequency f. In Example 1, the first option corresponds to this policy.

   (b) **Non-uniform allocation policy:** We synchronize elements at different rates. In particular, with a **proportional allocation policy** we synchronize element e_i at a frequency f_i that is proportional to its change frequency λ_i. Thus, the frequency ratio λ_i/f_i, is the same for any i under the proportional allocation policy. In Example 1, the second option corresponds to this policy.

3. **Synchronization order:** Now we need to decide in *what order* we synchronize the elements in the database.

   Example 2 We maintain a local database of 10,000 web pages from site A. In order to maintain the local copy up-to-date, we continuously update our local database by revisiting the pages in the site. In performing the update, we may adopt one of the following options:

symbol	meaning
(a) $\bar{F}(S)$, $\bar{F}(e_i)$	Freshness of database S (and element e_i) averaged over time
(b) $\bar{A}(S)$, $\bar{A}(e_i)$	Age of database S (and element e_i) averaged over time
(c) $\bar{F}(\lambda_i, f_i)$, $\bar{A}(\lambda_i, f_i)$	Freshness (and age) of element e_i averaged over time, when the element changes at the rate λ_i and is synchronized at the frequency f_i
(i) λ_i	Change frequency of element e_i
(j) f_i $(= 1/I_i)$	Synchronization frequency of element e_i
(k) λ	Average change frequency of database elements
(l) f $(= 1/I)$	Average synchronization frequency of database elements

Table 1: The symbols that are used throughout this paper and their meanings

- We maintain an explicit list of all URLs in the site, and we visit the URLs repeatedly in the same order. Notice that if we update our local database at a fixed rate, say 10,000 pages/day, then we synchronize a page, say p_1, at the fixed interval of one-day.
- We only maintain the URL of the root page of the site, and whenever we crawl the site, we start from the root page, following links. Since the link structure (and the order) at a particular crawl determines the page visit order, the synchronization order may change from one crawl to the next. Notice that under this policy, we synchronize a page, say p_1, at variable intervals. For instance, if we visit p_1 at the end of one crawl and at the beginning of the next crawl, the interval is close to zero, while in the opposite case it is close to two days.
- Instead of actively synchronizing pages, we synchronize pages on demand, as they are *requested* by a user. Since we do not know which page the user will request next, the synchronization order may appear random. Under this policy, the synchronization interval of p_1 is not bound by any value. It may range from zero to infinity. □

We can summarize the above options as follows:

(a) **Fixed order:** We synchronize all elements in the database in the *same* order repeatedly. Therefore, a particular element is synchronized at a *fixed interval* under this policy. This policy corresponds to the first option of the above example.

(b) **Random order:** We synchronize all elements repeatedly, but the synchronization order may be different in each iteration. This policy corresponds to the second option in the example.

(c) **Purely random:** At each synchronization point, we select an arbitrary element from the database and synchronize it. Therefore, an element is synchronized at intervals of arbitrary length. This policy corresponds to the last option in the example.

In Section 4 we will compare how effective these synchronization order policies are.

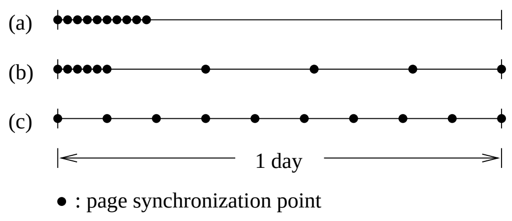


Figure 5: Several options for the synchronization points

4. **Synchronization points:** In some cases, we may need to synchronize the database only in a limited time-window. For instance, if a web site is heavily accessed during day-time, it might be desirable to crawl the site only in the night, when it is less frequently visited. We illustrate several options due to this constraint by an example.

   Example 3 We maintain a local database of 10 pages from site A. The site is heavily accessed during day-time. We consider several synchronization policies, including the following:

   - **Figure 5(a):** We synchronize all 10 pages in the beginning of the day, say midnight.
   - **Figure 5(b):** We synchronize most pages in the beginning of the day, but we still synchronize some pages during the rest of the day.
   - **Figure 5(c):** We synchronize 10 pages uniformly over a day. □

   In this paper, we assume that we synchronize the database uniformly over time. We believe this assumption is valid especially for the web environment. Because the web sites are located in many different time zones, it is not easy to identify which time zone a particular web site resides in. Also, the access pattern to a web site varies widely. For example, some web sites are heavily accessed during day time, while others are accessed mostly in the evening, when users are at home. Since crawlers cannot guess the best time to visit each site, they typically visit sites at a uniform rate that is convenient to the crawler.

policy	*Freshness* $\bar{F}(S)$	*Age* $\bar{A}(S)$
Fixed-order	$\frac{1-e^{-r}}{r}$	$I(\frac{1}{2}-\frac{1}{r}+\frac{1-e^{-r}}{r^2})$
Random-order	$\frac{1}{r}(1-(\frac{1-e^{-r}}{r})^2)$	$I(\frac{1}{3}+(\frac{1}{2}-\frac{1}{r})^2-(\frac{1-e^{-r}}{r^2})^2)$
Purely-random	$\frac{1}{1+r}$	$I(\frac{r}{1+r})$

Table 2: Freshness and age formula for various synchronization-order policies

4 Comparison of synchronization-order policies

Clearly, we can increase the database freshness by synchronizing more often. But exactly how often should we synchronize, for the freshness to be, say, 0.8? Conversely, how much freshness do we get if we synchronize 100 elements per second? In this section, we will address these questions by analyzing synchronization order policies. Through the analysis, we will also learn which synchronization-order policy is the best in terms of freshness and age.

In this section we assume that all real-world elements are modified at the same average rate λ. That is, we adopt the *uniform change-frequency* model (Section 2.3). When the elements change at the same rate, it does not make sense to synchronize the elements at different rates, so we also assume the *uniform allocation* policy (Item 2a in Section 3). These assumptions significantly simplify our analysis, while giving us solid understanding on the issues that we address.

Based on these assumptions, we analyze different synchronization-order policies in detail in [3], and we we summarize the result in Table 2. In the table, we use r to represent the frequency ratio λ/f, where λ is the frequency at which a real-world element changes and $f(=1/I)$ is the frequency at which a local element is synchronized. When $r < 1$, we synchronize the elements more often than they change, and when $r > 1$, the elements change more often than we synchronize them.

To help readers interpret the formulas, we show the freshness and the age graphs in Figure 6. In the figure, the horizontal axis is the frequency ratio r, and the vertical axis shows the freshness and the age of the local database. Notice that as we synchronize the elements more often than they change ($\lambda \ll f$, thus $r = \lambda/f \to 0$), the freshness approaches 1 and the age approaches 0. Also, when the elements change more frequently than we synchronize them ($r = \lambda/f \to \infty$), the freshness becomes 0, and the age increases. Finally, notice that the freshness is not equal to 1, even if we synchronize the elements as often as they change ($r = 1$). This result comes for two reasons. First, an element changes at random points of time, even if it changes at fixed *average* rate. Therefore, the element may not change between some synchronizations, and it may change more than once between other synchronizations. For this reason, it cannot be always up-to-date. Second, some delay may exist between the change of an element and its synchronization, so some elements may be "temporarily obsolete," decreasing the freshness of the database.

The graphs of Figure 6 have many practical implications. For instance, we can answer all of the following questions by looking at the graphs.

- **How can we measure how fresh the local database is?** By measuring how frequently the real-world elements change,[1] we can estimate how fresh the local database is. For instance, when the real-world elements change once a day, and when we synchronize the local elements also once a day ($\lambda = f$ or $r = 1$), the freshness of the local database is $(e-1)/e \approx 0.63$, under the fixed-order policy.

  Note that we derived the equations in Table 2 assuming that the real-world elements change at the *same* rate λ. Therefore, the equations may not be true when the real-world elements change at *different* rates. However, we can still interpret λ as the *average* rate at which the whole database change, and we can use the formulas as approximations. Later in Section 5, we derive exact formula when the elements change at different rates.

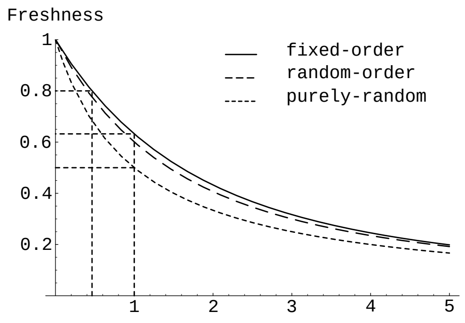
- **How can we guarantee a certain freshness of the local database?** From the graph, we can find how frequently we should synchronize the local elements in order to achieve a certain freshness. For instance, if we want at least 0.8 freshness, the frequency ratio r should be less than 0.46 (fixed-order policy). That is, we should synchronize the local elements at least $1/0.46 \approx 2$ times as frequently as the real-world elements change.

- **Which synchronization-order policy is the best?** The fixed-order policy performs best by both metrics. For instance, when we synchronize the elements as often as they change ($r = 1$), the freshness of the fixed-order policy is $(e-1)/e \approx 0.63$, which is 30% higher than that of the purely-random policy. The difference is more dramatic for age. When $r = 1$, the age of the fixed-order policy is only one fourth of the random-order policy. In general, as the variability in the time between visits increases, the policy gets less effective.

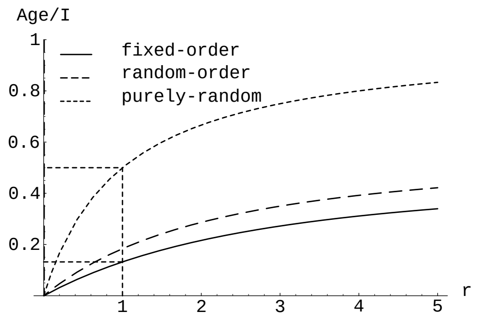
5 Comparison of resource-allocation policies

In the previous section, we addressed various questions, assuming that all elements in the database change at the same rate. But what can we do if the elements change at *different* rates and we know how often each element changes? Is it better to synchronize the element more often when it changes more often? In this section, we address this question by analyzing different resource-allocation policies (Item 2 in Section 3). For the analysis, we model the real-world database by the *non-uniform* change-frequency model (Section 2.3), and we

[1] In Section 6, we briefly discuss how we can measure the frequency of change. To learn more on this topic, please refer to [4].



(a) Freshness graph over $r = \lambda/f$



(b) Age graph over $r = \lambda/f$

Figure 6: Comparison of freshness and age of various synchronization policies

assume the *fixed-order* policy for the synchronization-order policy (Item 3 in Section 3), because the fixed-order policy is the best synchronization-order policy. In other words, we assume that the element e_i changes at the frequency λ_i (λ_i's may be different from element to element), and we synchronize e_i at the *fixed interval* $I_i(= 1/f_i,\ f_i$: synchronization frequency of e_i). Remember that we synchronize N elements in $I(= 1/f)$ time units. Therefore, the average synchronization frequency ($\frac{1}{N}\sum_{i=1}^{N} f_i$) should be equal to f.

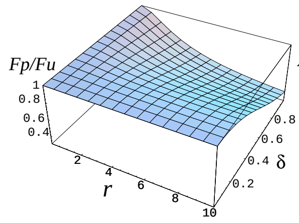
In Section 5.1, we start our discussion by comparing the uniform allocation policy with the proportional allocation policy. Surprisingly, the uniform policy turns out to be *always* more effective than the proportional policy. Then in Section 5.2 we try to understand why this happens by studying a simple example. Finally in Section 5.3 we study how we should allocate resources to the elements to achieve the optimal freshness or age.

5.1 Uniform and proportional allocation policy

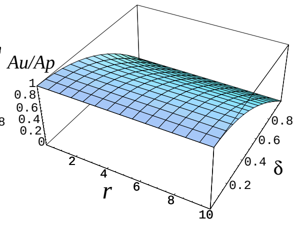
In this subsection, we first assume that change frequencies of real-world elements follow the *gamma distribution* and compare how effective the *proportional* and the *uniform* policies are. In [3], we prove that the conclusion of this section is valid for *any* distribution.

The gamma distribution is often used to model a random variable whose domain is non-negative numbers. Also, the distribution is known to cover a wide array of distributions. For instance, the exponential and the chi-square distributions are special instances of the gamma distribution, and the gamma distribution is close to the normal distribution when the variance is small. This mathematical property and versatility makes the gamma distribution a desirable one for describing the distribution of the change frequency.

Under these assumptions, we analyzed the uniform and proportional allocation policies for a database S [3], and we summarize the result in Table 3. In the table, r represents the frequency ratio λ/f, where λ is the average rate at which elements change (the mean of the gamma distribution), and f is the average rate at which we synchronize them ($1/I$). Also, δ represents the standard deviation of change frequencies (more precisely, $\delta^2 =$(variance)/(mean)^2 of the gamma distribution).



(a) $\bar{F}(S)_p/\bar{F}(S)_u$ graph over r and δ



(b) $\bar{A}(S)_u/\bar{A}(S)_p$ graph over r and δ

Figure 7: $\bar{F}(S)_p/\bar{F}(S)_u$ and $\bar{A}(S)_u/\bar{A}(S)_p$ graphs over r and δ

To help the discussion, we use the subscript p to refer to the proportional allocation policy and the subscript u to refer to the uniform allocation policy. Then, the uniform policy is better than the proportional one, when $\bar{F}(S)_p < \bar{F}(S)_u$ and $\bar{A}(S)_u < \bar{A}(S)_p$. To compare the two policies, we plot $\bar{F}(S)_p/\bar{F}(S)_u$ and $\bar{A}(S)_u/\bar{A}(S)_p$ graphs in Figure 7. Note that when the uniform policy is better, the ratios are below 1 ($\bar{F}(S)_p/\bar{F}(S)_u < 1$ and $\bar{A}(S)_u/\bar{A}(S)_p < 1$), and when the proportional policy is better, the ratios are above 1 ($\bar{F}(S)_p/\bar{F}(S)_u > 1$ and $\bar{A}(S)_u/\bar{A}(S)_p > 1$).

Surprisingly, we can clearly see that the ratios are below 1 for any r and δ values: The uniform policy is always better than the proportional policy! In fact, the uniform policy gets more effective as the elements change at more different frequencies. That is, when the variance of change frequencies is zero ($\delta = 0$), all elements change at the same frequency, so two policies give the same result ($\bar{F}(S)_p/\bar{F}(S)_u = 1$ and $\bar{A}(S)_u/\bar{A}(S)_p = 1$). But as δ increases (i.e., as the elements change at more different frequencies), $\bar{F}(S)_u$ grows larger than $\bar{F}(S)_p$ ($\bar{F}(S)_p/\bar{F}(S)_u \to 0$) and $\bar{A}(S)_u$ gets smaller than $\bar{A}(S)_p$ ($\bar{A}(S)_u/\bar{A}(S)_p \to 0$). Interestingly, we can observe that the age *ratio* does not change much as r increases, while the freshness *ratio*

allocation policy	*Freshness* $\bar{F}(S)$	*Age* $\bar{A}(S)$
Uniform	$\frac{1-(1+r\delta^2)^{1-\frac{1}{\delta^2}}}{r(1-\delta^2)}$	$\frac{I}{(1-\delta^2)}[\frac{1-\delta^2}{2} - \frac{1}{r} + \frac{1-(1+r\delta^2)^{2-\frac{1}{\delta^2}}}{r^2(1-2\delta^2)}]$
Proportional	$\frac{1-e^{-r}}{r}$	$\frac{I}{(1-\delta^2)}[\frac{1}{2} - \frac{1}{r} + \frac{1-e^{-r}}{r^2}]$

Table 3: Freshness and age formula for various resource-allocation policies

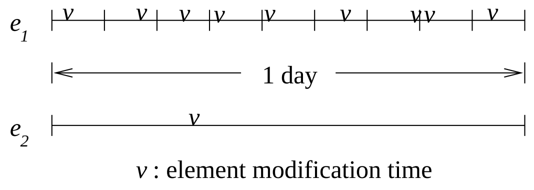


Figure 8: A database with two elements with different change frequency

heavily depends on the r value.

While we showed that the uniform policy is better than the proportional one only for the gamma distribution model, it is in fact very general conclusion. In [3], we prove that the uniform policy is *always* better than the proportional policy under *any* distribution.

5.2 Two element database

Intuitively, we expected that the proportional policy would be better than the uniform policy, because we allocate more resources to the elements that change more often, which may need more of our attention. But why is it the other way around? In this subsection, we try to understand why we get the unintuitive result, by studying a very simple example: a database consisting of two elements. The analysis of this simple example will let us understand the result more concretely, and it will reveal some intuitive trends. We will confirm the trends more precisely when we study the optimal synchronization policy later in Section 5.3.

Now we analyze a database consisting of two elements: e_1 and e_2. For the analysis, we assume that e_1 changes at 9 times/day and e_2 changes at once/day. We also assume that our goal is to maximize the freshness of the database averaged over time. In Figure 8, we visually illustrate our simple model. For element e_1, one day is split into 9 intervals, and e_1 changes *once and only once* in each interval. However, we do not know exactly when the element changes in one interval. For element e_2, it changes *once and only once* per day, and we do not know when it changes. While this model is not exactly a Poisson process model, we adopt this model due to its simplicity and concreteness.

Now let us assume that we decided to synchronize only *one* element per day. Then what element should we synchronize? Should we synchronize e_1 or should we synchronize e_2? To answer this question, we need to compare how the freshness changes if we pick one element over the other. If the element e_2 changes in the middle of the day and if we synchronize e_2 right after it changed, it will remain up-to-date for the remaining half of the day. Therefore, by synchronizing element e_2 we get 1/2 day "benefit"(or freshness increase). However, the probability that e_2 changes before the middle of the day is 1/2, so the "expected benefit" of synchronizing e_2 is $1/2 \times 1/2$ day $= 1/4$ day. By the same reasoning, if we synchronize e_1 in the middle of an interval, e_1 will remain up-to-date for the remaining half of the interval (1/18 of the day) with probability 1/2. Therefore, the expected benefit is $1/2 \times 1/18$ day $= 1/36$ day. From this crude estimation, we can see that it is more effective to select e_2 for synchronization!

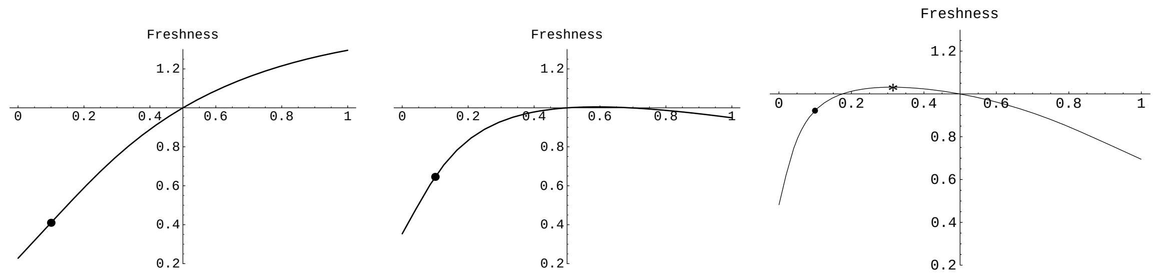
row	$f_1 + f_2$	f_1	f_2	benefit	best	
(a)	1	1	0	$\frac{1}{2} \times \frac{1}{18} = \frac{1}{36}$	0	1
(b)		0	1	$\frac{1}{2} \times \frac{1}{2} = \frac{9}{36}$		
(c)	2	2	0	$\frac{1}{2} \times \frac{1}{18} + \frac{1}{2} \times \frac{1}{18} = \frac{2}{36}$	0	2
(d)		1	1	$\frac{1}{2} \times \frac{1}{18} + \frac{1}{2} \times \frac{1}{2} = \frac{10}{36}$		
(e)		0	2	$\frac{1}{3} \times \frac{2}{3} + \frac{1}{3} \times \frac{1}{3} = \frac{12}{36}$		
(f)	5	3	2	$\frac{3}{36} + \frac{12}{36} = \frac{30}{72}$	2	3
(g)		2	3	$\frac{2}{36} + \frac{6}{16} = \frac{31}{72}$		
(h)	10	9	1	$\frac{9}{36} + \frac{1}{4} = \frac{36}{72}$	7	3
(i)		7	3	$\frac{7}{36} + \frac{6}{16} = \frac{41}{72}$		
(j)		5	5	$\frac{5}{36} + \frac{15}{36} = \frac{40}{72}$		

Table 4: Estimation of benefits for different choices

Table 4 shows the expected benefits for several other scenarios. The second column shows the total synchronization frequencies ($f_1 + f_2$) and the third column shows how much of the synchronization is allocated to f_1 and f_2. In the fourth column we estimate the expected benefit, and in the last column we show the f_1 and f_2 values that give the *highest* expected benefit. To save space, when $f_1 + f_2 = 5$ and 10, we show only some interesting (f_1, f_2) pairs. Note that since $\lambda_1 = 9$ and $\lambda_2 = 1$, row (h) corresponds to the proportional policy ($f_1 = 9, f_2 = 1$), and row (j) corresponds to the uniform policy ($f_1 = f_2 = 5$). From the table, we can observe following interesting trends:

1. **Rows (a)-(e):** When the synchronization frequency ($f_1 + f_2$) is much smaller than the change frequency ($\lambda_1 + \lambda_2$), it is better to give up synchronizing the elements that change too fast. In other words, when it is not possible to keep up with everything, it is better to focus on what we can track.

2. **Rows (h)-(j):** Even if the synchronization frequency is relatively large ($f_1 + f_2 = 10$), the uniform allocation policy (row (j)) is more effective than the proportional allocation policy (row (h)). The opti-



(a) $f_1 + f_2 = 1$ (b) $f_1 + f_2 = 3$ (c) $f_1 + f_2 = 10$

Figure 9: Series of freshness graphs for different synchronization frequency constraints. In all of the graphs, $\lambda_1 = 9$ and $\lambda_2 = 1$.

mal point (row (i)) is located somewhere between the proportional policy and the uniform policy.

We can verify this trend using our earlier analysis based on a Poisson process. We assume that the changes of e_1 and e_2 are Poisson processes with change frequencies $\lambda_1 = 9$ and $\lambda_2 = 1$. To help the discussion, we use $\bar{F}(\lambda_i, f_i)$ to refer to the time average of freshness of e_i when it changes at λ_i and is synchronized at f_i. Then, the freshness of the database is

$$\begin{aligned}\bar{F}(S) &= \frac{1}{2}(\bar{F}(e_1) + \bar{F}(e_2)) = \frac{1}{2}(\bar{F}(\lambda_1, f_1) + \bar{F}(\lambda_2, f_2)) \\ &= \frac{1}{2}(\bar{F}(9, f_1) + \bar{F}(1, f_2)).\end{aligned}$$

When we fix the value of $f_1 + f_2$, the above equation has only one degree of freedom, and we can plot $\bar{F}(S)$ over, say, f_2. In Figure 9, we show a series of graphs obtained this way. The horizontal axis here represents the fraction of the synchronization allocated to e_2. That is, when $x = 0$, we do not synchronize element e_2 at all ($f_2 = 0$), and when $x = 1$ we synchronize only element e_2 ($f_1 = 0$ or $f_2 = f_1 + f_2$). Therefore, the middle point ($x = 0.5$) corresponds to the uniform policy ($f_1 = f_2$), and $x = 0.1$ point corresponds to the proportional policy (Remember that $\lambda_1 = 9$ and $\lambda_2 = 1$). The vertical axis in the graph shows the *normalized* freshness of the database. We normalized the freshness so that $\bar{F}(S) = 1$ at the uniform policy ($x = 0.5$). To compare the uniform and the proportional policies more clearly, we indicate the freshness of the proportional policy by a dot, and the x and the y axes cross at the uniform policy.

From these graphs, we can clearly see that the uniform policy is always better than the proportional policy, since the dots are always below the origin. Also note that when the synchronization frequency is small (graph (a)), it is better to give up on the element that changes too often (We get the highest freshness when $x = 1$ or $f_1 = 0$). When $f_1 + f_2$ is relatively large (graph (c)), the optimal point is somewhere between the uniform policy and the proportional policy. The freshness is highest when $x \approx 0.3$ in Figure 9(c) (the star in the graph).

5.3 The optimal resource-allocation policy

From the previous discussion, we learned that the uniform policy is indeed better than the proportional policy. Also, we learned that the optimal policy is neither the uniform policy nor the proportional policy. For instance, we get the highest freshness when $x \approx 0.3$ for Figure 9(c). Then, what is the best way to allocate the resource to elements for a general database S? In this section, we will address this question. More formally, we will study how often we should synchronize individual elements when we know how often they change, in order to maximize the freshness or age. Mathematically, we can formulate our goal as follows:

Problem 1 Given λ_i's ($i = 1, 2, \ldots, N$), find the values of f_i's ($i = 1, 2, \ldots, N$) which maximize

$$\bar{F}(S) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(e_i) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda_i, f_i)$$

when f_i's satisfy the constraints

$$\frac{1}{N}\sum_{i=1}^{N} f_i = f \quad \text{and} \quad f_i \geq 0 \quad (i = 1, 2, \ldots, N)$$

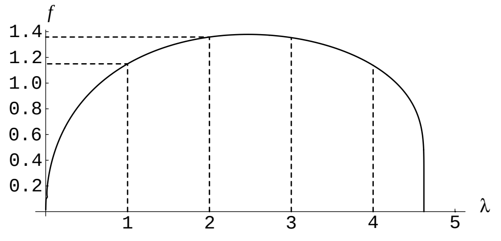
□

Because we can derive the closed form of $\bar{F}(\lambda_i, f_i)$,[2] we can solve the above problem by the *method of Lagrange multipliers* [12]. To illustrate the property of its solution, we use the following example.

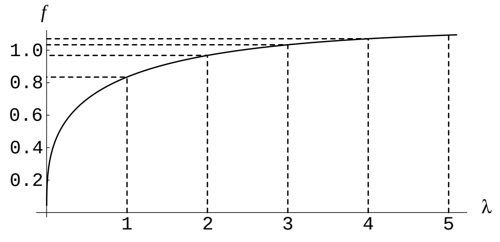
Example 4 The real-world database consists of five elements, which change at the frequencies of 1, 2, . . . , 5 (times/day). We list the change frequencies in row (a) of Table 5 (We explain the meaning of rows (b) and (c) later, as we continue our discussion.). We decided to synchronize the local database at the rate of 5 elements/day total, but we still need to find out how often we should synchronize each element.

For this example, we can solve the above problem numerically, and we show the graph of its solution

[2]For instance, $\bar{F}(\lambda_i, f_i) = (1 - e^{-\lambda_i/f_i})/(\lambda_i/f_i)$ for the fixed-order policy.



(a) change frequency vs. synchronization frequency for freshness optimization



(b) change frequency vs. synchronization frequency for age optimization

Figure 10: Solution of the freshness and age optimization problem of Example 4

	e_1	e_2	e_3	e_4	e_5
(a) change frequency	1	2	3	4	5
(b) synchronization frequency (freshness)	1.15	1.36	1.35	1.14	0.00
(c) synchronization frequency (age)	0.84	0.97	1.03	1.07	1.09

Table 5: The optimal synchronization frequencies of Example 4

in Figure 10(a). The horizontal axis of the graph corresponds to the change frequency of an element, and the vertical axis shows the optimal synchronization frequency of the element with that given change frequency. For instance, the optimal synchronization frequency of e_1 is 1.15 ($f = 1.15$), because the change frequency of element e_1 is 1 ($\lambda = 1$). Similarly from the graph, we can find the optimal synchronization frequencies of other elements, and we list them in row (b) of Table 5.

Notice that while e_4 changes twice as often as e_2, we need to synchronize e_4 less frequently than e_2. Furthermore, the synchronization frequency of e_5 is zero, while it changes at the highest rate. This result comes from the shape of Figure 10(a). In the graph, when $\lambda > 2.5$, f decreases as λ increases. Therefore, the synchronization frequencies of the elements e_3, e_4 and e_5 gets smaller and smaller. □

While we obtained Figure 10(a) by solving Example 4, we can prove that the shape of the graph is the same for *any* distributions of λ_i's [3]. That is, the optimal graph for *any* database S is *exactly the same* as Figure 10(a), except that the graph of S is scaled by a constant factor from Figure 10(a). Since the shape of the graph is always the same, the following statement is true in any scenario: *To improve freshness, we should penalize the elements that change too often.*

Similarly, we can compute the optimal *age* solution for Example 4, and we show the result in Figure 10(b). The axes in this graph are the same as before. Also, we list the optimal synchronization frequencies in row (c) of Table 5. Contrary to the freshness, we can observe that we should synchronize the element more often when it changes more often ($f_1 < \cdots < f_5$). However, notice that the difference between the synchronization frequencies is marginal: All f_i's are approximately close to one. In other words, the optimal solution is rather close to the uniform policy than to the proportional policy. Similarly for age, we can prove that the shape of the optimal age graph is always the same as Figure 10(b). Therefore, the trend we observed here is very general and holds for *any* database.

6 Experiments

Throughout this paper we modeled database changes as a Poisson process. In this section, we first verify the Poisson process model using experimental data collected from 270 sites for more than 4 months. Then, using the observed change frequencies on the web, we compare the effectiveness of our various synchronization policies. The experimental results will show that our optimal policy performs significantly better than the current policies used by crawlers.

6.1 Experimental setup

To collect the data on how often web pages change, we crawled around 720,000 pages from 270 "popular" sites every day, from February 17th through June 24th, 1999. This was done with the Stanford WebBase crawler, a system designed to create and maintain large web repositories. The system is capable of high indexing speeds (about 60 pages per second), and can handle relatively large data repositories (currently 300GB of HTML is stored). In this section we briefly discuss how the particular sites were selected for our experiments.

To select the sites for our experiment, we used the snapshot of the web in our WebBase repository. Currently, WebBase maintains the snapshot of 42 million web pages, and based on this snapshot we identified the top 400 "popular" sites as the candidate sites. To measure the popularity of sites, we essentially counted how many pages in our repository have a link to each site, and we used the count as the popularity measure of a site.[3]

Then, we contacted the webmasters of all candidate sites asking their permission for our experi-

[3]More precisely, we used PageRank as the popularity measure, which is similar to the link count. To learn more about PageRank, please refer to [10, 5].

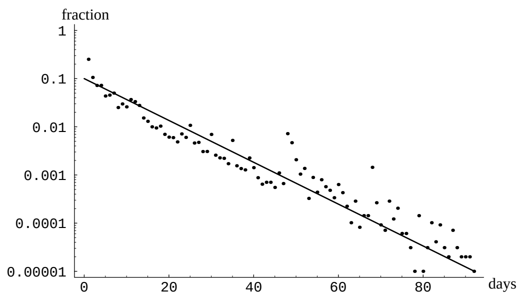


Figure 11: Change intervals for pages with the average change interval of 10 days

ment. After this step, 270 sites remained, including sites such as Yahoo (`http://yahoo.com`), Microsoft (`http://microsoft.com`), and Stanford (`http://www.stanford.edu`). Obviously, focusing on the "popular" sites biases our results to a certain degree, but we believe this bias is toward what most people are interested in.

From each site chosen this way, we selected around 3,000 pages and crawled them every day. From this daily update information, we can measure how often a page changes. For instance, when we detected 4 changes during our 4 month experiment, we can reasonably infer that the page changes every month on average. Later, we also briefly talk about the limitation of our experiment when we present the result of our experiment.

6.2 Verification of Poisson process

In this subsection, we verify whether the Poisson process adequately models web page changes. In Lemma 1, we computed how long it takes for a page to change under the Poisson process. According to the lemma, the time between changes follow the exponential distribution $\lambda e^{-\lambda t}$. We can use this result to verify our assumption. That is, if we plot the time between changes of a page p_i, the time should be distributed as $\lambda_i e^{-\lambda_i t}$, if changes of p_i follow a Poisson process.

In Figure 11, we show that the changes of a web page can indeed be modeled by the Poisson process. To plot this graph, we first selected only those pages whose *average* change intervals were 10 days and measured the time between changes in those pages. (We also plotted graphs for the pages with other average change intervals, and got similar results when we had sufficient data.) From this data we could get the distribution of the change intervals, which is shown in Figure 11. The horizontal axis represents the interval between changes, and the vertical axis shows the fraction of changes with that interval. The vertical axis in the graph is logarithmic to emphasize that the distribution is exponential. The line in the graph is what a Poisson process would predict. While there exist small variations, we can clearly see that Poisson process predicts very well the observed data.

While this result strongly indicates that a Poisson process is a good model for the web page changes, our result is also limited. Since we crawled pages on a daily basis, we could not obtain detailed change histories for the pages that change very often, and because we conducted our experiment only for 4 months, we could not detect changes to the pages that rarely change. Also, there may exist a set of pages that are updated at regular intervals, which may not necessarily follow a Poisson process.

However, we believe that it is safe to use the Poisson model for the following reasons. First, crawlers rarely can visit a page every day,[4] so most crawlers do not particularly care exactly how often a page changes if the page changes very often (say, more than once every day). Also, when the crawler manages hundreds of millions of pages, it is very difficult to identify the pages that are updated regularly, so we may assume that the set of pages managed by the crawler are modified by a random process *on average.*

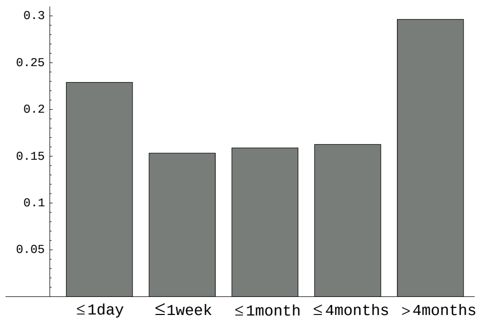


Figure 12: Percentage of pages with given average interval of change

6.3 Frequency of change and its implication

Based on the data that we collected, we report how many pages change how often, in Figure 12. In the figure, the horizontal axis represents the average change interval of pages, and the vertical axis shows the fraction of pages changed at the given average interval. For instance, we can see that about 23% of pages changed more than once a day from the first bar of Figure 12.

From this data, we can estimate how much improvement we can get, if we adopt the optimal-allocation policy. For the estimation, we assume that we maintain 100 million pages locally and that we synchronize all pages every month.[5] Also based on Figure 12, we assume that 23% of pages change every day, 15% of pages change every week, etc. For the pages that did not change in 4 months, we assume that they change every year. While it is a crude approximation, we believe we can get some idea on how effective different policies are.

In Table 6, we show the predicted freshness and age for various resource-allocation policies. To compute the numbers, we assumed the fixed-order policy (Item 3a in Section 3) as the synchronization-order policy. We can

[4]Crawlers should not abuse web sites. Otherwise, the site administrators sometimes block accesses.

[5]Many popular search engines report numbers similar to these.

policy	*Freshness*	*Age*
Proportional	0.12	400 days
Uniform	0.57	5.6 days
Optimal	0.62	4.3 days

Table 6: Freshness and age prediction based on the real web data

clearly see that the optimal policy is significantly better than any other policies. For instance, the freshness increases from 0.12 to 0.62 (500% increase!), if we use the optimal policy instead of the proportional policy. Also, the age decreases by 23% from the uniform policy to the optimal policy. From these numbers, we can also learn that we need to be very careful when we optimize the policy based on the frequency of change. For instance, the proportional policy, which people may intuitively prefer, is significantly worse than any other policies: The age of the proportional policy is 100 times worse than that of the optimal policy!

7 Related work

References [5] and [2] also study how to improve a web crawler. However, these references focus on how to *select* the pages to *initially* crawl, in order to improve the "quality" of the local collection. Contrary to these works, we studied how to *maintain* the collection up-to-date. Reference [6] studies how to schedule the web crawler to improve the freshness. The model used for web pages is similar to ours; however, the model for the crawler and freshness is very different. In data warehousing context, a lot of work has been done to efficiently maintain the local copy, or the *materialized view* [7, 8, 13]. However, most of the work focused on different issues, such as minimizing the size of the view while reducing the query response time [8].

8 Conclusion

In this paper we studied how to synchronize a local database to improve its freshness and age. We presented a formal framework, which provides a theoretical foundation for this problem, and we studied the effectiveness of various refresh policies. In our study we identified a potential pitfall (proportional synchronization), and proposed an optimal policy that can improve freshness and age very significantly. Finally, we investigated the changes of real web pages and validated our analysis based on this experimental data.

In our current framework, we assumed that for users the freshness or age of every element is equally important. But what if the elements have different "importance"? For example, if the database S consists of two elements (e_1 and e_2), and if e_1 is twice as important as e_2 ($F(S) = \frac{1}{3}[2F(e_1) + F(e_2)]$), how should we synchronize them to maximize the freshness? While we need more thorough analysis to answer this question, our preliminary result indicates that we need to synchronize e_1 more often than e_2, but not necessarily twice as often.

As more and more digital information becomes available, it will be increasingly important to collect it effectively. A crawler or a data warehouse simply cannot refresh all its data constantly, so it must be very careful in deciding what data to poll and check for freshness. The policies we have studied in this paper can make a significant difference in the "temporal quality" of the data that is collected.

Acknowledgement

We thank Vasilis Vassalos for his thoughtful comments at the early stage of this work. We also thank Chris Jihye Won for her constant support and encouragement.

References

[1] Google Inc. `http://www.google.com`.

[2] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *The 8th International World Wide Web Conference*, 1999.

[3] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. Technical report, Stanford University, 1999. `http://www-db.stanford.edu/~cho/papers/cho-synch.ps`.

[4] J. Cho and H. Garcia-Molina. Estimating frequency of change. Technical report, Stanford University, 2000.

[5] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. *Computers networks and ISDN systems*, 30:161–172, 1998.

[6] E. Coffman, Jr., Z. Liu, and R. R. Weber. Optimal robot scheduling for web search engines. Technical report, INRIA, 1997.

[7] J. Hammer, H. Garcia-Molina, J. Widom, W. J. Labio, and Y. Zhuge. The Stanford data warehousing project. *IEEE Data Engineering Bulletin*, June 1995.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD Conference*, 1996.

[9] S. Lawrence and C. L. Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[10] L. Page and S. Brin. The anatomy of a large-scale hypertextual web search engine. *Computers networks and ISDN systems*, 30:107–117, 1998.

[11] H. M. Taylor and S. Karlin. *An Introduction To Stochastic Modeling.* Academic Press, 3rd edition, 1998.

[12] G. B. Thomas, Jr. *Calculus and analytic geometry.* Addison-Wesley, 4th edition, 1969.

[13] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *ACM SIGMOD Conference*, 1995.