Scalable Application-Aware Data Freshening

Donald Carney
Brown University

Sangdon Lee
Mokpo National University

Stan Zdonik
Brown University

Abstract

Distributed databases and other networked information systems use copies or mirrors to reduce latency and to increase availability. Copies need to be refreshed. In a loosely coupled system, the copy sites are typically responsible for synchronizing their own copies. This involves polling and can be quite expensive if not done in a disciplined way. This paper explores the topic of how to determine a refresh schedule given knowledge of the update frequencies and limited bandwidth. The emphasis here is on how to use additional information about the aggregate interest of the user community in each of the copies in order to maximize the perceived freshness of the copies. This paper develops a model and an optimal solution for small cases, presents several heuristic algorithms that work for large cases, then explores the impact of object size on the refresh schedule. It also presents experimental evidence that our algorithms perform quite well.

1 Introduction

Replication is a well-known data management technique for improving the performance and availability of a distributed computing system. Internet mirror sites geographically distribute the load for popular information sources. Data warehouses replicate data from the transactional database in order to provide a platform for decision support queries that does not interfere with the originating source.

Of course, as updates occur at the underlying data sources, the replicas can drift out of synch. In tightly coupled distributed systems, algorithms exist for atomically installing an update to all replicas. In more loosely coupled systems like the Internet, heavy-weight synchronization schemes are not feasible. Instead methods that respect the independence of the participants are much more common. For example, a site that stores copies may itself be responsible for detecting changes in the underlying source and for updating its local copy accordingly.

This work was supported by the National Science Foundation under NSF grant number IIS00-86057 and by the postdoctoral fellowships program of the Korea Science & Engineering Foundation (KOSEF)

While a good solution might involve an active server that pushes updates to the mirror, most existing data providers do not support this. Without cooperation from the source, the site with a local copy must poll the source in order to detect changes. If this is done too often with respect to the update frequency, resources are wasted. If it does not occur often enough, data freshness suffers. The goal, then, is to pick an appropriate polling frequency for each local copy that balances these two needs.

Data management has always benefited when we can use application-level knowledge to make decisions. For example, database indices are built on attributes that are known to be commonly used. In modern networked environments such as the Internet, there is evidence that more responsive systems can be delivered if users supply a description of their interests (a.k.a. a *profile*). Consider MyYahoo and Tivo, the personal video recording system. In these systems, users supply profiles to create personalized content; however, personalization is only one application of profiles. In the case of Tivo, the profile is also used as a way to manage the local cache in the Tivo box. This notion can be extended to a wide range of data management situations. In particular, in this paper, we show that profiles can be used effectively to control replica management policies.

Others [5] have studied techniques for managing the process of updating local copies. In these studies, the goal has been to maximize the average freshness of a collection of copies. This is accomplished by producing a schedule for polling master copies to see if they have changed. In this paper, we show that there is much to be gained from taking user interests into account when producing a schedule for freshening. Intuitively, it makes little sense to spend freshening resources on items that no one wants.

The contributions of our work are as follows:

- a model for refreshing local copies that takes user interest into account.
- heuristic techniques that allow our model (and previously studied models) to scale.
- an extension that considers variable sized objects.

Section 2 describes our first contribution and sets up the problem in more precise terms including a definition

of our fundamental metric, *perceived freshness*. Section 3 describes the first part of our second contribution. It shows why refreshing solutions don't scale and addresses the problem of designing a practical algorithm for handling large cases, analyzes the problem using our new metric, presents results for the optimal case, and shows why this solution doesn't scale. Sections 4 discusses our experimental framework as well as our experimental results. Section 5 explores the impact of object size. Section 6 describes related work, and Section 7 concludes with a summary of our results and some future directions.

2 Perceived Freshness

We are concerned with the problem of keeping a set of copies as up to date as possible with respect to a master data source. We will speak of the set of copies as the *mirror* and the master data source as the *source*. A single member of the mirror will be referred to as a *local copy*. The computer that is responsible for maintaining the currency of the mirror will be called the *mirror site*. Intuitively, the *freshness* of a mirror is the fraction of its local copies whose value reflects the current value in the source. It is important to note that copies are not added or deleted at the mirror.

The mirror site is connected to the source with a fixed and limited bandwidth. The limitation in bandwidth will determine how many refresh operations can be performed per unit time. The work in this paper is concerned with planning the best use of this bandwidth by determining a refresh schedule that will maximize the freshness of the mirror.

Our work follows closely on the work presented in [5]. Thus, we briefly summarize their notion of freshness for a local database S consisting of N elements ($S = e_1, \cdots, e_N$). It should be noted that as in [5], we initially assume all objects have the same fixed size and, therefore, take up the same amount of space in the mirror. In Section 5, we relax this assumption to determine the impact of object size.

Definition 1 *The freshness of a local element e_i at time t is*

$$F(e_i;t) = \begin{cases} 1 & \text{; if } e_i \text{ is up-to-date at time } t \\ 0 & \text{; otherwise} \end{cases}$$

Definition 2 *The freshness of a local database S at time t*

$$F(S;t) = \frac{1}{N}\sum_{i=1}^{N} F(e_i;t)$$

Freshness of an element is binary. At any instant, the element at the mirror is either up-to-date (1) or not (0). The freshness of a local database at any instant is the average freshness of all elements at the local database at that instant.

The freshness metrics given above make perfect sense for the case in which no knowledge of the access patterns exists. After all, in the absence of profiles, it is reasonable to assume that all accesses to local copies are equally likely. However, if all accesses are not equally likely, we can define a freshness metric that weights the contribution of each local copy by its popularity (expected number of accesses per unit time). We call our metric *perceived freshness*. Perceived freshness captures what users see as they use the mirror. *Average Perceived Freshness* measures the freshness that users observe as they access various portions of the mirror. If a given item is never accessed, it does not contribute to the average perceived freshness regardless of how stale its value is.

We assume that many users frequently access the mirror site. In order to improve their chances of getting fresh data, they each submit a *profile*. Simply stated, a profile is a declarative specification of the relative importance of each copy in the mirror. This could be written in a sophisticated profile language [3]; however, for our purposes, we assume that the profile is a distribution of access frequencies. If the access frequencies are scaled by the total number of accesses per unit time, the profile can be viewed as an access probability distribution. The assumption is that a user will access important items more often than unimportant items. This is, of course, not in general required, but it simplifies our discussion.

The mirror collects all the user profiles and aggregates them into one master profile that is a combined frequency distribution for all users. Our algorithms make decisions based on the master profile. Thus, we use the term *profile* in what follows, to refer to the master profile.

It is important to note that our profile model is a very simple one. We assume that user interest in an element x is directly proportional to the frequency with which that user accesses x. This is clearly not always the case. We can imagine more sophisticated profiles [3], but find it useful for the purposes of this study to restrict our attention to a limited, but plausible model. It should also be noted that it is not difficult to modify our profile model to be a density function over any measurable attribute of our objects. For example, in a stock market application, a profile might be cast as a plot of importance vs. ticker symbol. Also, individual profiles can be weighted before they are aggregated into the master profile, so as to give higher priority to more important users (e.g., generals or higher paying customers).

2.1 Problem Definition

Suppose that over some period of time the mirror experiences a set of M accesses called the *access set* that we denote as $A = a_1, \cdots, a_M$. A given a_j names an element e_i in the mirror. The access set A can contain more than one access to a given element. If we denote the number of accesses for an element e_i as M_i, then $M = \sum_{i=1}^{N} M_i$. It follows that each element e_i has an access probability, $p_i = \frac{M_i}{M}$ (note that $\sum_{i=1}^{N} p_i = 1$).

While average overall freshness is an interesting system

metric, it is not that relevant to individual users. A user might rate the quality of the mirror by "keeping score" at each access. If the user gets a fresh copy, score a point. If the user gets a stale copy, score nothing. In this way, the score would reflect how well the system is freshening those items that user actually references. *Perceived freshness* captures this idea.

Let $F(a_i;t)$ be the freshness of the element referenced by access a_i. We define *perceived freshness* as follows.

Definition 3 *The perceived freshness of a set of accesses A at time t is*

$$PF(A;t) = \frac{1}{M}\sum_{i=1}^{M} F(a_i;t)$$

Intuitively, $PF(A;t)$ is the fraction of accesses that see an up-to-date copy.

Definition 4 *The time averaged perceived freshness of a set of local database accesses A is*

$$\overline{PF}(A) = \lim_{t\to\infty}\frac{1}{t}\int_o^t PF(A;t)dt$$

The time averaged perceived freshness for a mirror S that experiences an access set A is $\overline{PF}(S)$. It is this last quantity, $\overline{PF}(S)$, that is important to our algorithms. From the definitions, we can prove that $\overline{PF}(S) = \overline{PF}(A) = \sum_{i=1}^{N} p_i\overline{F}(e_i)$ where $\overline{F}(e_i)$ is the time-averaged freshness of element e_i. (For a detailed proof, refer to [2]).

We assume that it is possible to obtain the number of updates to an element over some time period. Prior work has shown how the source can use estimation [4] and sampling [6] techniques to obtain a good estimate of these update frequencies. These frequency estimates would be periodically communicated to the mirror.

Given these definitions and assumptions, we can now formulate our problem more precisely.

Core Problem Given an update frequency λ_i for each element at the source and an access probability p_i for each of the local copies in the mirror (i = 1,2,..., N), find the refresh frequencies f_i (i = 1,2,..., N) which maximize

$$\overline{PF}(S) = \sum_{i=1}^{N} p_i\overline{F}(e_i) = \sum_{i=1}^{N} p_i\overline{F}(f_i,\lambda_i) \qquad (1)$$

satisfying the following constraint

$$\sum_{i=1}^{N} f_i = TotalBandwidth \quad \text{and} \quad f_i \geq 0 (i = 1, 2, ..., N) \qquad (2)$$

In equation (1), we write $\overline{F}(f_i,\lambda_i)$ to represent the time averaged freshness for e_i given its synchronization frequency f_i and its change rate λ_i.

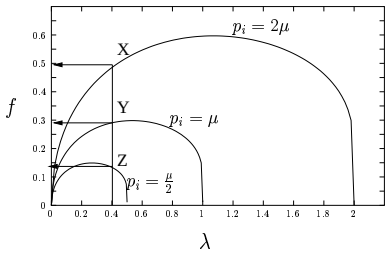


Figure 1. Relationship among f, λ, p

Note that the bandwidth is given in terms of the number of refreshes that are allowed over some time period. The refresh frequencies f_i are computed as the number of refreshes of local copy i that should be scheduled over the same time period.

2.2 The Optimal Solution

In this section, we examine the optimal solution to the core problem.

Solving the core problem produces a bandwidth allocation for each local copy. Given such an allocation, we must further determine how those allocations should be ordered in time. In [5], a Fixed Order synchronization-order policy is shown to give the best performance. In this policy, all objects are synchronized in the same order repeatedly, each at a fixed interval. Since we can determine a closed form of $p_i\overline{F}(\lambda_i, f_i)$, it is possible to solve the core problem by the method of Lagrange multipliers. We show the general solution of the problem in the Appendix.

To solve the problem we must solve an optimization problem with a non-linear objective function and with a linear constraint on the overall available bandwidth. This problem is solved using non-linear programming to produce a schedule with a fixed synchronization rate for each local copy. Such a solution will optimally produce a schedule that maximizes the freshness of the mirror given the constraint on bandwidth. Since the complexity of non-linear programming is bounded by a high-order polynomial, this approach is feasible for small sets of objects only.

From the solution, we can produce the graph in Figure 1. We can understand the relationships among f_i, λ_i and p_i better by examining the graphs for various p_i's. Figure 1 shows three curves on which solutions are located respectively for objects with access probabilities $p_i = \{\frac{\mu}{2}, \mu, 2\mu\}$. Again, we assume the Fixed Order refresh policy. It clearly shows that for any given change rate (λ) an element e_i needs to get more bandwidth as its access probability p_i increases. For example, a point Y on the graph for $p_i = \mu$ moves to point X as p_i is increased to $p_i = 2\mu$. Again, starting from point Y, as p_i is decreased to half its original value, we move to point Z. However, note that increasing p_i means that some other $p_j (i \neq j)$ needs to be decreased since $\sum p_i = 1$. Elements with large p_i will get higher f_i

than elements with lower p_i and the same λ. In Figure 1, an element with $\lambda = 1.2$ does not get any bandwidth when $p_i = \mu$. However, it will require significant bandwidth as its p_i becomes 2μ. This suggests the general principle that elements with large λ_i need increasing bandwidth as their p_i's increase.

2.2.1 Some Intuition

Here we illustrate the characteristics of the solution for perceived freshness versus the solution given in [5] for average freshness which ignores user interest. We will base these observations on a toy example that was used in [5].

Example The database consists of five equal-sized elements, which change at the frequencies of 1,2,$\cdots$, 5 (times/day). Let's assume that there are 3 sets of access probabilities for these five elements as follows. P1 = (1/5, 1/5, 1/5, 1/5, 1/5), P2= (1/15, 2/15, 3/15, 4/15, 5/15) and P3= (5/15, 4/15, 3/15, 2/15, 1/15). P1 represents a uniform access distribution for the five elements, while P2 and P3 represent skewed access distributions. P2 represents the case for which the hottest elements change most frequently. P3 represents the opposite case. We assume that freshening the local database can only occur at a rate of 5 elements/day (the bandwidth constraint). We can solve this example problem numerically to determine the best freshening schedule.

Table 1 shows the results. Row (a) in table 1 shows change frequencies for each element. Row (b) shows the optimal synchronization frequencies for the uniform distribution case (P1) which coincide with the results in [5]. However, as rows (c) and (d) show, optimal synchronization frequencies for non-uniform access distributions are much different from the result for the uniform distribution case. Especially in row (c), for which the most frequently changing element (e_5) now needs to get the highest synchronization frequency (instead of 0 as in row (b)). This implies that frequently changing elements may need to be synchronized much more than less frequently changing elements if they are more likely to be accessed. This case, where users are more interested in items which change frequently, is not uncommon in practice (for example, volatile stocks might be more interesting to day-traders purely due to their volatility). In the case of less frequently changing elements, row (d) shows that they should receive more freshening bandwidth if the access frequency for them is high. □

We will show how differences in optimal synchronization frequencies such as those illustrated above will affect the perceived freshness in Section 2.2.2.

2.2.2 Solving Optimally for Small Cases

We can solve the synchronization problem optimally for a small number of objects. Table 2 shows the setup we use. As in [5] we assume a gamma distribution for the distribution of change frequencies of database objects. We use a

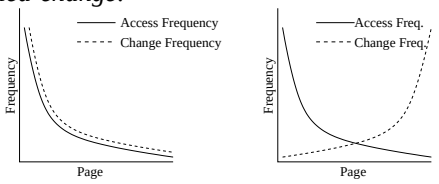
Table 1. Optimal sync frequencies for example

	e_1	e_2	e_3	e_4	e_5
(a)change freq	1	2	3	4	5
(b)sync freq (P1)	1.15	1.36	1.35	1.14	0.00
(c)sync freq (P2)	0.33	0.67	1.00	1.33	1.67
(d)sync freq (P3)	1.68	1.83	1.49	0.00	0.00

Zipfian [1] distribution to model user profiles from which we generate a master profile of user interest. It has been shown in practice that θ can be as high as 1.6 [17]. The example we use is a database with 500 objects and each object is updated on average twice during a synchronization period (the per interval update rate of individual objects is determined by the gamma distribution - the mean of the gamma is 2). We provide a sensitivity analysis study of these parameters in [2]. We assume no relationship between these parameters and object size, and, for now, assume that all objects have the same size (we relax this restriction later).

In this section the GF technique refers to General Freshening (proposed in [5]) which optimizes average freshness and the PF technique refers to Perceived Freshening (our approach). In both cases, we measure the average perceived freshness achieved with the resulting schedule. While it is not too surprising that the Perceived Freshening technique does better when you measure perceived freshness, it is interesting to note exactly where the benefits lie.

We chose three possible alignments of change and access frequency. Figure 2 shows two of these configurations. Note that the shapes of the curves in the figure are for demonstration purposes and that their actual shapes would be determined by their distributions. In the *aligned* case, objects that change frequently are of highest interest and in the *reverse* case, volatile objects are less interesting to users. A third case shuffles the change rates from the gamma distribution randomly across the elements so that no relationship exits between interest and updates. We call this *shuffled-change*.



(a) aligned (b) reverse

Figure 2. Alignment options

Figures 3(a)-3(c) shows the perceived freshness results. The graphs all share the same characteristic: when the

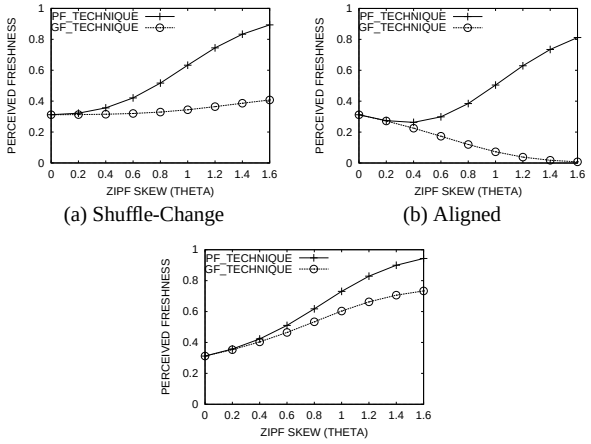
[1] A Zipfian distribution is commonly used to model non-uniform data access. As its parameter θ increases, the access pattern becomes increasingly skewed. The probability of accessing any object i is proportional to $(1/i)^{\theta}$.

users' interests are considered in the synchronization algorithm, the users' perceived freshness increases as users' interests become more skewed. Note that in all cases, when $\theta = 0$, perceived freshness equals general freshness. This is because $\theta = 0$ corresponds to a uniform distribution. Both synchronization techniques produce the exact same schedule because all data items have the same access probability.

As expected, the PF technique outperforms the GF technique as the interest skew increases. The most significant difference is the aligned case in which perceived freshness approaches 0 for high interest skew when user interest is ignored.

Table 2. Setup for Ideal Experiments

NumObjects	500
NumUpdatesPerPeriod	1000
NumSyncsPerPeriod	250
Theta (θ)	0.0 - 1.6
UpdateStdDev(δ)	1.0



(a) Shuffle-Change (b) Aligned

(c) Reverse

Figure 3. The Ideal Case for Perceived Freshness and General Freshness

3 Scalability and Heuristic Approaches

While the optimization of *freshness* and *perceived freshness* presented in section 2 are interesting for studying the effectiveness of refresh algorithms, they are not accurately computable in practice for large problems.

The problem is classified as a convex optimization problem with a non-linear objective function[2]. It is known that convex optimization problems have polynomial complexity. However, it is still not scalable for a huge number of variables. Even for a convex optimization problem with a linear objective function, the worst case complexity is $O(n^{3.5})$ where n is the number of variables [10]. For non-linear objective functions the complexity is worse. Empirically, we have observed that running our non-linear programming package is manageable for problems with fewer than a thousand items; however, as we increase the size of the problem to hundreds of thousands of items, the package runs for days without terminating.

Managing mirrors with millions of elements is common in many real-world applications such as web search engines or data warehouses. For large real-world problems for which the contents of the mirror or the user interests might change, we would need to periodically solve the Core Problem to ensure that the freshening schedule still produces good results. The time needed to solve for the optimal result will be intolerable. Previous work [5] does not consider this. In this section, we will explain several heuristic algorithms that reduce the problem size. We will also show that, for the most part, the accuracy of these algorithms compares favorably to the optimal solution.

The heuristic approaches we present in this section consist of two steps. The first step divides elements in a database into G partitions such that the number of elements in each of G partitions is $T_1, T_2, \cdots, T_G$ and $\sum_{i=1}^{G} T_i = N$.

In the second step, we pick a representative element from each of the partitions and solve the optimization problem for each of these G elements. This solution allocates bandwidth to each partition. This bandwidth is divided evenly among the elements of that partition.

The quality of the solution will depend on how the partitions are constructed and how the original optimization problem is transformed to a smaller more manageable problem. In the following, we describe our set of alternatives for partitioning and for constructing the smaller optimization problem.

3.1 Partitioning Alternatives

We define four partitioning techniques. Each of our techniques creates partitions in the same way. All N elements are sorted. Then N/G successive elements are assigned to a partition. When N is divisible by G, the number of elements in each partition will be equal. If N is not divisible by G, some of the partitions will have fewer elements. The effect on the performance from this difference will be negligible for very large N and relatively small G, which will be the usual case.

Clearly, p and λ will have an effect on the result, thus, we consider each of them as well as two ways of combining them as our sorting criteria.

p-Partitioning In this approach, all N elements in the database are sorted by access frequency (p) and therefore, elements in each partition will have similar access probabilities.

[2] We can verify that $p_i\overline{F}(\lambda_i, f_i)$ is convex over f_i by computing second order derivatives of the analytic form for the Fixed Order policy.

λ-Partitioning This approach is similar to P-Partitioning except that the elements in the database are sorted by change frequency (λ). This technique is included for completeness.

$\frac{p}{\lambda}$-Partitioning This partitioning alternative orders elements by (p/λ). The intuition behind this approach comes from the fundamental results found in [5] and in section 2. As change rate increases, synchronization frequency allocation should decrease and second, as access probability increases, synchronization frequency allocation should increase. Quantity $\frac{p}{\lambda}$ reflects these properties.

PF-Partitioning Though the partitioning techniques described above can identify similarities between elements based on their characteristics (p and λ) they do not reflect the relative importance of access probability and change frequency on perceived freshness. The PF-Partitioning approach orders elements by their perceived freshness (PF) given a fixed synchronization frequency [3]. The exact synchronization frequency used in our calculations is not important. We use a synchronization frequency of 1.0.

3.2 Optimization

After the partitioning phase, we need to solve an optimization problem to determine the best synchronization frequencies. We first treat all elements in a partition as identical. For identical elements, synchronization frequencies will be the same. The optimization problem on groups of identical elements is transformed to an equivalent problem in which the same objective function (but over fewer elements) and bandwidth constraint operate on a data set that includes one representative from each partition. This yields a bandwidth allocation for each partition which, then, must be allocated proportionately to each element in that partition. In this approach, optimization can be done in one step.

A second approach would transform the original problem into a number of smaller problems, in which only a small number of elements participate. Then G optimization problems are solved, each with a small number of elements. This approach requires multiple steps. We considered this approach but it does not make sense for large problems because it is still very costly to run. Whereas solving each individual subproblem is feasible, the sheer number of subproblems is too large. For example, if it is tolerable (solvable within an hour) to solve the optimization problem over 1000 elements, you would have to solve 1000 such problems for a database with 1,000,000 elements. Thus we did not report the results of these multi-stage algorithms.

To be clear, we formalize our optimization step. For G partitions, partition i ($i \leq G$) is regarded as having T_i identical elements. A representative must be chosen that reflects the similarity among these elements. For each partition C_i with T_i elements, a representative element is determined by averaging the access probabilities and the change frequencies of all elements in the partition. Thus, a representative element v_i for partition i has access probability(p_i) and change frequency(λ_i) as follows.

$$p_i = \sum_{e_j \in \text{ partition i}} p_j/T_i, \qquad \lambda_i = \sum_{e_j \in \text{ partition i}} \lambda_j/T_i$$

The original optimization problem is converted to a smaller problem with G representative elements as shown below. Note that the objective function and constraint for the original problem have been changed by scaling the terms by the number of elements in each partition.

- **Transformed Problem** Given λ_i's and p_i's (i = 1,2,..., G) for each representative element, find the values of f_i's(i = 1,2,..., G) which maximize

$$\overline{PF}(S) = \sum_{i=1}^{G} p_i \overline{F}(f_i, \lambda_i) T_i$$

when f_i's satisfy the constraint

$$\sum_{i=1}^{G} T_i f_i = TotalBandwidth \quad \text{and} \quad f_i \geq 0$$

Solving for f_i gives us an amount of bandwidth that is assigned to each of the elements of partition i.

4 Experiments

In this section we describe and analyze performance results for large numbers of elements for which we use our approximation techniques.

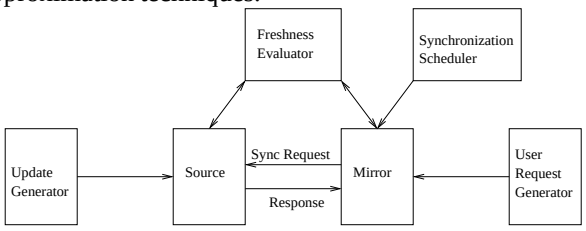


Figure 4. Simulation Model

A detailed simulation was built to demonstrate various synchronization algorithms. Figure 4 shows the model for the simulation. The parameters for the simulator are described below.

- **NumObjects:** number of objects in the database. In the current simulation, all objects have the same size.
- **PeriodLength:** length of a sync period in simulation clock time
- **NumSyncsPerPeriod:** number of syncs per period (can be characterized as the sync bandwidth)

[3] When we use the Fixed-order synchronization policy, $PF(e_i) = p_i \times \frac{1-e^{-\lambda_i/c}}{\lambda_i/c}$ where c is the fixed synchronization frequency.

- **UpdateStdDev(δ):** standard deviation of update distribution (gamma)
- **Theta(θ):** skew of the access distribution (zipf)

The *Synchronization Scheduler* has knowledge of the master profile and object change frequencies. From these it creates the schedule of synchronization requests for the *Mirror*. We use IMSL [9] libraries to solve the non-linear objective function.

The *Freshness Evaluator* operates in two modes. It can analytically calculate freshness metrics based on simulated access and update workloads, or it can track system activity by monitoring updates and user requests and computing the metrics. The results shown in the next section have been verified using both modes.

4.1 Heuristic Case - Comparing the Techniques

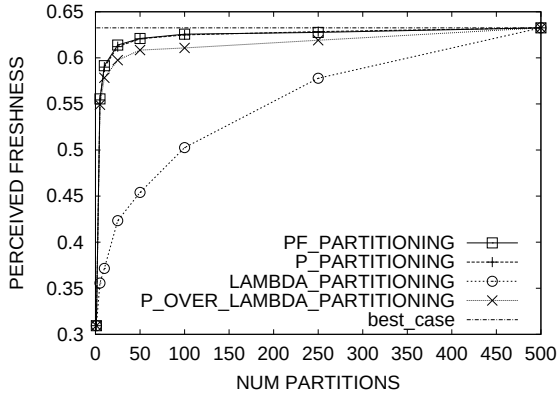
In this section we show the results from our simulator for the approximation techniques described in section 3. We first compare the techniques to each other and to the optimal case for relatively small problem sizes. Next we demonstrate the differences between the techniques for much larger cases for which solving optimally is infeasible. We also show empirically that our solutions scale.

We compare techniques based on the quality of the result and the scalability of the technique.

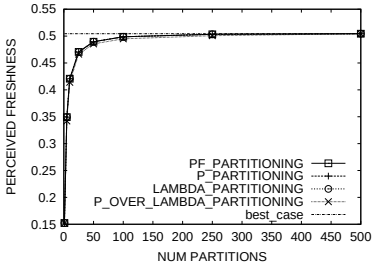
4.1.1 Quality of the Result

We use the set of parameters specified in Table 2 to compare the heuristic approaches. Recall from section 2.2.2 we study three alignment cases: *aligned*, *reverse*, and *shuffled-change*. Figure 5 shows the relative performance of our four partitioning techniques for these cases. With any of these techniques, as the number of partitions increases, the number of items in each partition decreases, and the approximate solution naturally approaches the ideal solution. The plots verify this and show that as the number of partitions increases the solution grows closer to the ideal case. The astute reader will notice that there is little difference between the techniques in Figures 5(b) and 5(c). This is because the partitioning techniques for the *aligned* and *reverse* alignments produce nearly identical results. In the *aligned* case, it is obvious that the partitions would be roughly the same for p-Partitioning and λ-Partitioning (both distributions have similar shape). Since access probability dominates the Perceived Freshness equation, PF-Partitioning and p-Partitioning generate very similar partitions for the case where the access is skewed. In both alignment cases, $\frac{p}{\lambda}$-Partitioning will order based on whichever parameter is more highly skewed, therefore it follows that $\frac{p}{\lambda}$-Partitioning will be nearly the same as the other three.

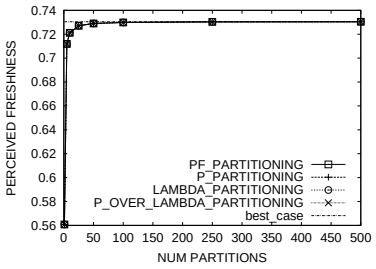
Hereafter we will only consider the *shuffled-change* alignment for comparing partitioning techniques. We do this because the independence of p and λ presents more of an average case rather than the extrema represented by *aligned* and *reverse* alignments and the average case demonstrates the differences between the partitioning techniques more plainly.



(a) Shuffle Change



(b) Aligned



(c) Reverse

Figure 5. Comparing the Techniques

Examination of the graph in Figure 5(a) reveals that some techniques approach the optimal solution more quickly (with a smaller number of partitions) than others under *shuffled-change* alignment . Note that p, PF, and $\frac{p}{\lambda}$ Partitioning approach the optimal solution more quickly than λ Partitioning. As we had seen from the optimal case in section 2.2.2, Perceived Freshness is dominated by access frequency. Figure 6 shows this more clearly. In the figure, the λ Partitioning technique can not achieve the same level of perceived freshness as θ is increased. Also notice that the plots have the same general shape as those for Perceived Freshness in Figure 3. The plot shows that as θ increases, perceived freshness increases for all of the techniques. The reason for this is that for high θ a small number of the ele-

ments that have a high access probability receive the bulk of synchronization bandwidth. For low θ, elements with high change frequencies only have modest interest, therefore, λ dominates the impact on Perceived Freshness.

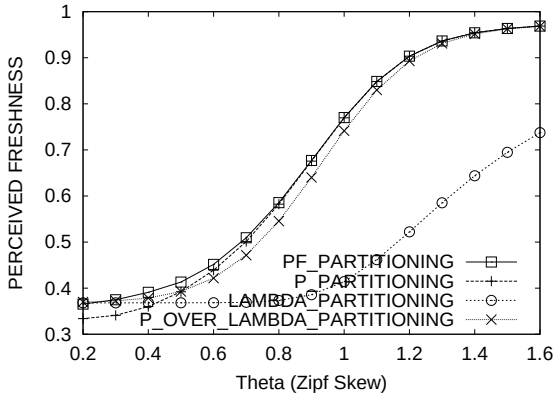


Figure 6. Sensitivity of Partitioning to Zipf Skew (θ) in Shuffle-Change Alignment

4.1.2 Scalability of the Techniques

All of the approximation techniques we have studied share the same basic principle: divide the search space into pieces and solve. The time required to solve the problem is based on the technique used to divide the search space and the resultant optimization problem(s). For large cases, the goal is to use the smallest number of partitions to achieve a good approximate answer.

Table 3. Setup for Partitioning Experiments

NumObjects	500000
NumUpdatesPerPeriod	1000000
NumSyncsPerPeriod	250000
Theta (θ)	1.0
UpdateStdDev(δ)	2.00

Table 3 shows the parameter setup used to demonstrate the performance of the two techniques. Figure 7 shows plots similar to those in figure 5, however, the size of the problem is much larger. Though we cannot verify the ideal solutions for the cases shown here, the shape of the graph suggests the results from previous runs.

Notice that PF-PARTITIONING is still the clear winner and that the solutions using more than 100 partitions do not appreciably improve the answer.

4.1.3 An Additional Improvement

Empirically we have seen that the partitions can be improved by running several iterations of a k-Means clustering algorithm. In this technique, we start with partitions (clusters) as before, and we use k-Means clustering to clean up any clustering problems in our initial partitions. The distance metric we use for clustering is Euclidean Distance as described below:

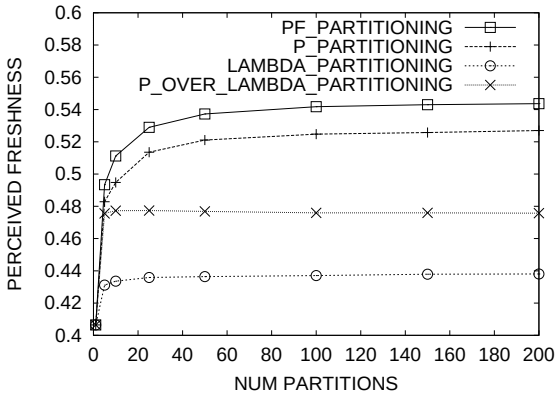


Figure 7. Big Case

- **Euclidean Distance** The Euclidean distance between two elements, $e_1(p_1, \lambda_1)$ and $e_2(p_2, \lambda_2)$, is defined as follows. Note that λ_i's are normalized into $\check{\lambda}_i$'s so that $\sum_{i=1}^{N} \check{\lambda}_i = 1$ and $0 \leq \check{\lambda}_i \leq 1$.

$$E_{dist}(e_1, e_2) = \sqrt{(p_1 - p_2)^2 + (\check{\lambda_1} - \check{\lambda_2})^2} \quad (3)$$

During each iteration of k-Means clustering, each element is compared with the mean (average p and λ) of each partition (or cluster) and moved to the closest one. With each successive iteration, a better clustering is obtained. See Figure 8. In each iteration of the clustering, there are $N \times k$ comparisons for N elements and k clusters. Thus, it is possible to improve the result in two different ways. We can increase the number of partitions, or we can run more iterations of k-Means clustering to improve the quality of the partitions. Figure 9 demonstrates this tradeoff. In the figure, the x-axis is time, and the thick line in the figure labeled "CLUSTER LINE" contains the points for the "0 iterations" case in Figure 8. That is, if you changed the x-axis of Figure 8 to time, the "0 iterations" plot would correspond to the "CLUSTER LINE" plot in figure 9. The other plots in Figure 9 show how Perceived Freshness improves using the k-Means clustering step. The first point on each of those plots (the point on "CLUSTER LINE") is 0 iterations, the next successive for each plot is 1 iteration followed by 3, 5, 7, 10, 15, and 25 iterations (when they fall within range on the graph). The very interesting phenomena is that with very few iterations, significant gains are seen in Perceived Freshness when clustering is used to improve the partitions created by PF-Partitioning. For this particular setup, a good solution is found earliest with 50 partitions and 10 iterations which finishes in 62 seconds.

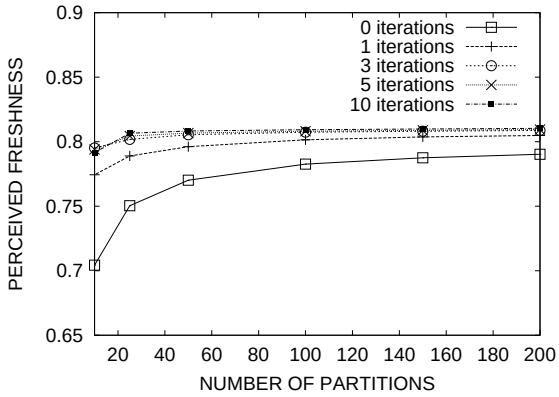


Figure 8. Improvement in Perceived Freshness after Clustering

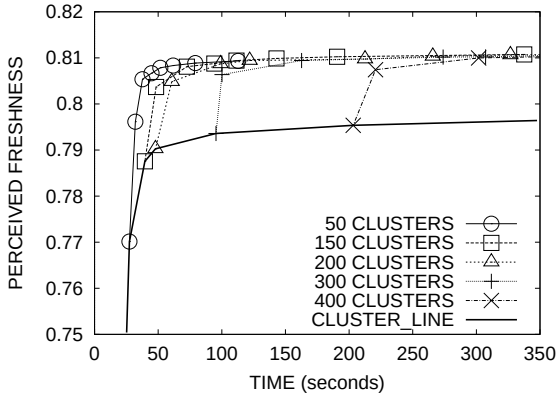


Figure 9. Improvement in Perceived Freshness after Clustering

5 Object Size Considerations

5.1 Problem Extension

We now relax the fixed object size assumption. Clearly, freshening a large object will take more bandwidth than a small object. For example, if an object whose size is 3 units of bandwidth is synchronized 2 times during a period, it will now consume 6 units of bandwidth. Recall from the earlier discussion of the Core Problem that for fixed size objects the linear constraint requires the sum of the refresh frequencies to be equal to the total bandwidth available (equation 2). For variable sized objects, this constraint becomes the sum of the products of sync frequencies (f_i) and object sizes (s_i) must be less than or equal to the total bandwidth (equation 4).

$$\sum_{i=1}^{N} s_i f_i \leq TotalBandwidth \quad \text{and} \quad s_i \geq 0, f_i \geq 0 \quad (4)$$

5.2 Freshening Approaches

Since the extended problem is different from the core problem only in its constraint specification, the optimal freshening problem can be solved with little difficulty.

For the freshening of a large number of variable-sized elements where heuristics are needed, we take the same steps of partitioning and optimization as described in section 3. In the partitioning step, we have to change the calculation of Perceived Freshness for PF-Partitioning. When we do not consider object size, we assume a constant amount of bandwidth for each element (see footnote 3). When considering object size, it is not fair to assume a constant bandwidth, because one sync of a big page costs more bandwidth than one sync of a small page. Therefore, we divide the constant bandwidth by the object size. We call this PFS-Partitioning.

The optimization alternatives are basically the same except that the constraint now becomes $\sum_{i=1}^{G} T_i \overline{s_i} f_i = TotalBandwidth$. The average of the size of elements in the same partition is represented as $\overline{s_i}$.

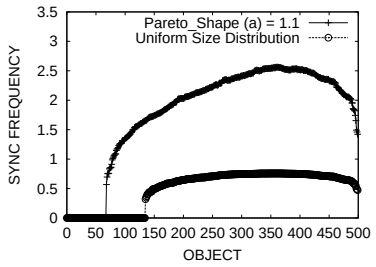
5.3 Results and Further Analysis

Other studies [12] have shown that object size on the web tends to follow a Pareto[4] distribution. Figure 10 shows the optimal allocation of synchronization resources for two different page size distributions (uniform, where each object has the same size (1.0) and Pareto, where the object sizes follow a Pareto and have a mean of 1.0). For simplicity, access is uniform ($\theta = 0.0$) and change rate is aligned (object 1 has the highest change rate). Page size is also aligned (object 1 is largest). Figure 10(a) shows the number of synchronizations given to each item. Figure 10(b) shows the amount of synchronization bandwidth allocated to each item. Because the Pareto case has a large number of small objects, the total number of syncs is larger while the total amount of synchronization bandwidth is the same. It is important to note that when object size is variable, a smaller object can be given more synchronizations while consuming less bandwidth. For both cases, notice that all of the sync resources are given to the pages with the lowest change rates.

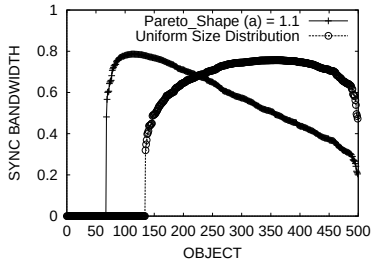
For large cases, there are important considerations to take into account concerning bandwidth allocation. There are two possible approaches for bandwidth allocation to the elements in the same partition:

1. **Fixed Refresh Frequency Allocation(FFA)** : All the elements get the same frequency of refresh. This technique was appropriate when all elements were of the

[4] A *Pareto* distribution typically captures the highly variable size of objects, where $F(x) = (k/x)^a, x \geq k$ a is the *shape* and k is the *scale*. The distribution has a mean of $ka/(a-1)$.



(a) Sync Frequency



(b) Sync Bandwidth

Figure 10. Optimal Sync Resource Distribution

same size. However, now this leads to disproportional bandwidth allocation. To address this we introduce the following technique.

2. **Fixed Bandwidth Allocation(FBA)** : The same amount of bandwidth is allocated for every element in the same partition. They will have different freshening frequencies which are determined by (assigned bandwidth[5]) / s_i. As a result, smaller objects will get higher number of refreshes than larger objects although they are in the same partition.

Figure 11 shows the relative performance of the allocation approaches described for PFS-Partitioning. For the experiment, the alignments of change rate and object size are reversed, and access is shuffled. (object 1 has a high change rate and a low size). This scenario seems to make sense for the real world. For instance, on the web, large objects like images and movies rarely change, whereas small objects like stock quotes and weather reports change quite often.

Notice that Fixed Bandwidth Allocation (FBA) approaches a better solution earlier (with fewer partitions) than with the Fixed Frequency Allocation (FFA). We have seen through empirical study of these allocation policies for small and large setups that FBA always outperforms FFA.

We have also applied the clustering improvement described in section 4.1.2 to the object size case and have found that it again improves the solutions within a small

[5]Note that the bandwidth for a representative element for a partition i from the optimization step will be $\overline{s_i} \times f_i$.

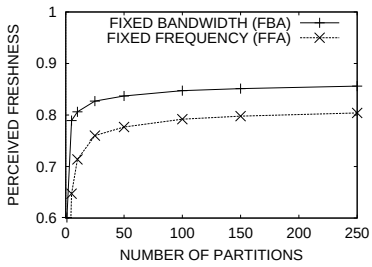


Figure 11. Sync Allocation

number of iterations [6]. We also implemented $SIZE$-Partitioning which orders elements by size. Like p and λ-Partitioning, ordering by size only does not capture the relationship between elements so as to improve Perceived Freshness as much as PFS-Partitioning.

To summarize, in the experiments of (fig. 10(a)), the synchronization frequencies that are produced when we ignore object size (uniform distribution) yield a perceived freshness of 0.312. Whereas the synchronization frequencies that are produced when we take the size distribution into account (Pareto) yield a perceived freshness of 0.586. As shown in fig. 10(b) when we ignore object size, too much bandwidth is given to large objects which robs bandwidth from smaller objects, thereby, making suboptimal use of bandwidth. Furthermore, in our heuristic algorithms, the bandwidth allocation policy is effected by the variation in object size. Objects should be given a fixed bandwidth allotment.

6 Related Work

There has been a great deal of work on managing replicas in distributed computing systems [18, 11, 8]. Much of this work requires careful cooperation and standard protocols between the participating sites in order to produce the appearance of perfectly synchronized replicas. Because of the autonomy of sites as well as severe performance requirements, many networked applications cannot support this level of cooperation. Thus, copies are allowed to drift out of synchronization and a freshening system like ours will periodically intervene to pull them back into agreement.

In data warehousing, much work has been done on managing *materialized views*[20] efficiently. However, the major focus of most of this work is on issues such as minimizing view size or reducing the view update time.

Data freshening shares some similarities with data caching in that they both must deal with the problem of out of date local copies. The problem of managing the freshness of local copies has been explored in the web caching area[19]. Here, however, the model of synchronization is

[6]Euclidean Distance between two elements is $E_{distSize}(e_1, e_2) = \sqrt{(p_1 - p_2)^2 + (\breve{\lambda_1} - \breve{\lambda_2})^2 + (\breve{s_1} - \breve{s_2})^2}$. The s_i's are normalized so that $\sum_{i=1}^{N} \breve{s_i} = 1$.

based on freshening actions that happen on explicit data requests from the clients or by updates pushed from server.

Recent work has addressed the maintenance of local copies for environments where exact consistency is not possible due to the limited resources. For *pull-based* approaches, clients decide on a refresh schedule based on knowledge of the change frequency of documents. This information can be obtained using Time-To-Live(TTL)[7] information, application of a probabilistic distributions[4], or sampling from servers[6]. A TTL represents the estimated period a web document will remain fresh and is widely used for web cache consistency. One study [4] discusses how frequency of change can be modeled by a poison process and can be estimated from observed data.

In [5], a freshness metric and synchronization policies are suggested and discussed. The work reported here extends their work. Their goal was to maximize the average freshness of a collection of copies. However, their work did not consider the user's access patterns. As a result, their freshening approach may not produce the most productive solution from the point of view of what a user actually sees. Further, their approach is not scalable for a very large number of data items, which could be impractical in many applications.

Sampling-based refresh policies [6] initially poll some portion of the documents from each server to determine the relative ratio of the changed documents in the samples. Based on the ranking of this ratio, a greedy method that refreshes documents starting from the highest ranked server produces the best result.

Furthermore, our analysis and simulation results reveal that access probability dominates change frequency when the access distribution is highly skewed which is typical in the Internet. As the access probability can be obtained easily at the mirror site, our approach is applicable even in the case with imperfect knowledge of change frequency.

Profiles can be more specific as in [1] where a refresh policy using clients' preferences about the recency and latency is known. This work does not consider limited bandwidth. Refresh is done by explicit client request. Our approach could be combined with theirs to provide a more aggressive refresh approach.

When data sources can notify clients of changes, clients can either determine the exact time to pull the changed documents[13], or data sources can *push* the changes to clients[16]. Update propagation strategies for the materialized WebView[13] have been proposed thereby determining the WebView update schedule based on exact knowledge of when objects are updated at the servers. [16] proposes a *push-based* refresh approach in which clients and servers cooperate with each other to determine refresh schedules dynamically based on variable bandwidth. However, there are many environments including the Web where this push-based approach can not be applied because data sources do not inform clients of changes.

In order to gain fine-grained control over the tradeoff between precision and performance, adaptive refresh on replicated data have been studied[15, 14]. The primary goal of this work is to control the amount of refresh over replicated approximate data to satisfy given data precision constraints. The client informs the server about what level of precision it requires, and the server sends the client a new value when the precision interval is violated.

7 Conclusions

We have presented a new criterion called *perceived freshness* for optimizing the freshness of a mirror given limited bandwidth. Unlike previous studies, we take into account the access frequencies for each copy.

We analyzed the optimal solution to the problem of maximizing perceived freshness, but observed that the solution for large problems take far too much time. For cases in which we need to revisit the refresh schedule often (e.g., frequent updates to the mirror), the high cost of the optimal solution is unacceptable. Thus, we study several heuristic partitioning techniques that produce approximate solutions at drastically reduced costs. The *PF-Partitioning* technique is shown to perform best and we show that it tracks the ideal solution for modest problem sizes. We further show that for very large problems, our techniques can produce good results in reasonable solution times.

The most surprising result was that reclustering our partitions before running the optimization algorithm had dramatic effects on the results at a small cost in running time. Thus, using a small number of simple partitions as a starting point for reclustering was shown to be the best approach.

Next we analyze the impact of object size on the problem and the approaches. We show the important distinction between synchronization frequency and synchronization bandwidth. We also present how to proportionally allocate synchronization bandwidth across different sized objects.

We believe that these results are relevant to many Internet-scale applications. They depend on the ability to gather information on user access-patterns, but we believe that this is feasible through direct feedback from users or from a simple learning algorithm that monitors the system request log.

Notice that in Figure 10 there are a significant number of objects that do not get refreshed at all. Thus, they get arbitrarily out of date and therefore become much less valuable. It would be interesting to investigate how space could be better used. For example, this could influence which objects we include in the mirror when the mirror is smaller than the database. Furthermore, we feel that this is an early example of what we call profile-driven data management, an approach that makes data management decisions in a

wide-area setting based on user profiles. This area holds promise for introducing data management ideas into a distributed setting without the need for a human DBA.

References

[1] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *VLDB*, 2002.

[2] D. Carney, S. Lee, and S. Zdonik. Profile-driven data freshening. Technical report, CS Dept., Brown University, 2002.

[3] M. Cherniack, M. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personnal Communication Magazine : Special Issue on Pervasive Computing*, August 2001.

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

[5] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the ACM SIGMOD*, pages 117–128, May 2000.

[6] Junghoo Cho and Alexandros Ntoulas. Effective change detection using sampling. In *VLDB*, 2002.

[7] Adam Dingle and Thomas Partl. Web cache coherence. In *Proceedings of the 5th International WWW Conference*, May 1996.

[8] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems (TOCS)*, 4(1), February 1986.

[9] IMSL, Inc. *IMSL C Numerical Libraries, Version 3.01*. Visual Numerics, 1998.

[10] N. K. Karmarkar. A new polynomial–time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[11] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

[12] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, May 2001.

[13] Alexandros Labrinidis and Nick Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB*, pages 391–400, 2001.

[14] C. Olston, B. Thau Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD*, pages 355–366, May 2001.

[15] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the VLDB*, pages 144–155, September 2000.

[16] Chris Olston and Jennifer Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD*, 2002.

[17] V. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implicatins. In *SIGCOMM*, pages 111–123. ACM, 2000.

[18] S. H. Son. Replicated data management in distributed database systems. *ACM SIGMOD Record*, 17(4), November 1988.

[19] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 25(9):36–46, October 1999.

[20] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD*, pages 316–327, May 1995.

Appendix: Solution for the Core Problem

Here we address the solution of the Core Problem as expressed by equations (1) and (2).

The perceived freshness of database S

$$\overline{PF}(S) = \sum_{i=1}^{N} p_i \overline{F}(e_i) = \sum_{i=1}^{N} p_i \overline{F}(f_i, \lambda_i)$$

takes its maximum, when all f_i's satisfy the equations

$$\frac{\partial p_i \overline{F}(f_i, \lambda_i)}{\partial f_i} = \mu \quad \text{and} \quad \sum_{i=1}^{N} f_i = TotalBandwidth$$

The solution consists of $(N + 1)$ equations (N equations of $\partial p_i \overline{F}(\lambda_i, f_i)/\partial f_i = \mu$ and one equation of $\sum_{i=1}^{N} f_i = TotalBandwidth$) with (N+1) unknown variables $(f_1, f_2, ..., f_N, \mu)$ by introducing another variable μ, which is a typical way to apply method of Lagrange multipliers. We can solve these $(N + 1)$ equations for the f_i's, since we know the closed form of $p_i \overline{F}(\lambda_i, f_i)$. From the solution, we can see that all optimal f_i's satisfy $\partial p_i \overline{F}(f_i, \lambda_i)/\partial f_i = \mu$.

To compute the optimal f_i for each λ_i and p_i, we need to solve the equation

$$\partial p_i \overline{F}(f_i, \lambda_i)/\partial f_i = \mu \tag{5}$$

for f_i. When the Fixed-order policy is applied, $\overline{F}(f_i, \lambda_i)$ is derived as $\frac{1-e^{-\lambda_i/f_i}}{\lambda_i/f_i}$. Unfortunately, Equation 5 is a non-linear equation which does not have a closed-form solution. Moreover, the equation has a parameter μ, whose value depends on the distribution of λ_i's and p_i's. Therefore, we may get totally different solution f for different distributions of λ_i's and p_i's. Cho et al.[5] proved that the solutions of Equation 5 are essentially the same, independently of μ when p_i's are equal.

Equation 5 can be rewritten as

$$\partial \overline{F}(f_i, \lambda_i)/\partial f_i = \frac{\mu}{p_i} \tag{6}$$

Equation 6[7] implies that a solution for an element $e_i(p_i, \lambda_i)$ which satisfy $\partial p_i \overline{F}(f_i, \lambda_i)/\partial f_i = \mu$ is on the graph of $\partial \overline{F}(f_i, \lambda_i)/\partial f_i = \frac{\mu}{p_i}$. When such $p_i = \mu$, the solution will be on the graph of $\partial \overline{F}(f_i, \lambda_i)/\partial f_i = 1$.

[7] For the Fixed-order policy, this becomes $\frac{1}{f} e^{-\lambda_i/f_i} + \frac{1}{\lambda_i}(1 - e^{-\lambda_i/f_i}) = \frac{\mu}{p_i}$ after applying partial derivation.