Adaptive Pull-Based Data Freshness Policies for Diverse Update Patterns

Laura Bright
OGI/OHSU
Beaverton, OR 97006
bright@cse.ogi.edu

Avigdor Gal
Technion - IIT
Haifa 32000 Israel
avigal@ie.technion.ac.il

Louiqa Raschid
University of Maryland
College Park MD 20742
louiqa@umiacs.umd.edu

ABSTRACT

An important challenge to effective data delivery in wide area environments is maintaining the data freshness of objects using solutions that can scale to a large number of clients without incurring significant server overhead. Policies for maintaining data freshness are traditionally either push-based or pull-based. Push-based policies involve pushing data updates by servers; they may not scale to a large number of clients. Pull-based policies require clients to contact servers to check for updates; their effectiveness is limited by the difficulty of predicting updates. Models to predict updates generally rely on some knowledge of past updates. Their accuracy of prediction may vary and determining the most appropriate model is non-trivial. In this paper, we present an adaptive pull-based solution to this challenge. We first present several techniques that use update history to estimate the freshness of cached objects, and identify update patterns for which each technique is most effective. We then introduce adaptive policies that can (automatically) choose a policy for an object based on its observed update patterns. Our proposed policies improve the freshness of cached data and reduce costly contacts with remote servers without incurring the large server overhead of push-based policies, and can scale to a large number of clients. Using trace data from a data-intensive website as well as two email logs, we show that our adaptive policies can adapt to diverse update patterns and provide significant improvement compared to a single policy.

1. INTRODUCTION

An important challenge to effective data delivery in wide area environments is maintaining the data freshness of local copies of objects using solutions that can scale to a large number of clients without incurring significant server overhead. In between client connections, object copies may become stale due to updates at the origin server. The challenge of keeping cached data fresh with respect to updates at remote servers despite periodic connectivity has been studied considerably in many different contexts, e.g., web caching ([5, 9, 12, 18]), mobile computing, and caching at database-driven websites ([3, 16]). At the heart of the problem is the inability of cache managers to accurately predict when the next update will occur. Therefore, either servers must notify caches of updates (push-based), or caches must contact servers to validate cached objects before delivering them to clients (pull-based). Using push-based policies, servers can provide guarantees with respect to the freshness of the data in a client's cache. However, these guarantees come at the cost of increased server overhead, e.g., keeping information on clients and the content of their cache. Push-based solutions can work well on fixed networks when the number of caches is relatively small, e.g., caching at a reverse proxy of a website that delivers dynamic content [3, 16]. However, they may not scale to a large number of clients.

In contrast, pull-based policies do not require servers to store any information about clients. Instead, clients or servers typically use heuristics to estimate how long the object will remain fresh in the cache. If an object is requested after this time expires, the cache must contact the remote server to check for updates. The advantage of pull-based policies is their straightforward implementation. However, the effectiveness of pull-based policies is limited by the difficulty of estimating the freshness of cached objects. Inaccurate estimates may cause the cache to validate objects that are still fresh (i.e., objects that have not been updated), or to serve stale data from the cache, both of which may reduce the benefits of caching. Research reported in [9] shows that as many as 30-50% of cache hits may result in unnecessary validations (freshness misses), so reducing the number of validations could significantly reduce access latencies and improve the effectiveness of caching. For mobile clients, excessive validations are particularly costly because they consume wireless bandwidth and battery power. However, these clients may require fresh data in many cases. Thus, an important challenge for caching with limited connectivity is how to keep cached data fresh with minimal contact with remote servers.

Existing heuristics to estimate the freshness of cached objects typically rely on update histories. The pull-based policy that is most commonly used in practice is Time-to-Live (TTL), which estimates an expiration time for an object as a function of the time it was last modified. This policy assumes that objects that were recently updated in the past are more likely to be updated in the near future, and does not consider earlier updates to an object. The TTL policy works well for objects that experience bursts of updates that occur close together. However, this policy may not work well for objects with more regular and predictable behavior.

Other works, e.g., [6, 7, 17] have proposed using more detailed information of past updates to an object to predict when it is likely to be updated in the future. These techniques use update histories to estimate the frequency of updates to an object. Such techniques may work better than TTL for objects with cyclic behavior. However, the use of update histories to estimate the freshness of objects is nontrivial. For example, an object may be more likely to be updated in the morning than in the afternoon or evening, so its update probability would vary at different times of day. On the other hand, if there is limited history available for an individual object, the object may experience updates that cannot be accurately predicted by its history. Thus, there are many challenges to effectively using object update histories. There is a need for techniques that can better exploit update history, as well as a need for solutions that are able to choose the most appropriate technique.

In this paper, we present a flexible approach to improve pull-based policies that relies on exploiting update histories of objects. We first introduce two techniques, IndHist and AggHist, that use update histories to estimate the freshness of cached objects. IndHist uses the update history of an individual object and can capture the behavior of a single object when sufficient history is available. AggHist uses an aggregate history aggregated over multiple objects with similar behavior, modeled as a *recurrent piecewise constant model* [11] and provides an approximation of an individual object's update history. It is useful when there is insufficient history for a single object. It also reduces storage and computational overhead for servers. We compare IndHist and AggHist to the widely used TTL [5, 12] and identify situations where each policy is most effective.

A key challenge to the effective exploitation of update histories is choosing the most appropriate technique to estimate the freshness of a given object. We present adaptive policies that can choose among IndHist, AggHist, and TTL according to client needs and object behavior. This enables cache managers to improve the accuracy of freshness estimates of cached objects by choosing the technique that is likely to perform best based on available history, *and requires no a priori classification of an object's update patterns.* An adaptive policy can offer significant improvements over using any one of the techniques alone.

This paper makes two contributions. First, we show that IndHist and AggHist can improve the accuracy of clients' estimates of the freshness of cached objects compared to TTL (for all TTL parameter settings) for many objects with cyclic update patterns. This, in turn, reduces the number of cached objects that need to be validated, thus reducing access latencies as well as the number of messages between caches and servers. Thus, for cyclic objects, we exploit history to reduce the number of contacts with servers. In comparison existing approaches to reduce the number of freshness misses [9] require validating objects offline and thus *increase* the total number of messages between clients and servers. Second, we introduce a class of adaptive policies that can choose among policies according to the behavior of individual objects. These policies have the advantage of adapting to individual objects, without requiring any a priori classification of the object's update pattern. We show the following key results:

- For objects with cyclic behavior, using history can reduce up to 60% of validations while providing comparable freshness to TTL for all TTL parameter settings.
- Aggregate history over multiple objects provides a good approximation when individual history is insufficient, and has reduced storage overhead.
- Adaptive policies can choose policies appropriate to individual objects based on update patterns and can improve on using a single policy. Further, they require no a priori classification of object behavior. These policies can also be tuned to meet preferences of diverse clients.
- We show that an adaptive IndHist/AggHist outperforms either IndHist or AggHist alone for objects with a cyclic update pattern, and an adaptive IndHist/TTL outperforms either IndHist or TTL for objects with a bursty update pattern. Cyclic and bursty patterns are described in Section 3.

Our policies have the advantage of being completely pull-based. Servers piggyback update history information on their responses to client requests, so servers need not push any information to clients or store any information about clients. Our solution does not require changing web servers or underlying protocols. Rather, it provides new technique for servers to improve the freshness of data copies and reduce the number of validation requests they receive.

The paper is organized as follows: Section 2 surveys related work. We classify update patterns and policies for modeling them in Section 3. We present several pull-based policies using update histories in Section 4, and compare these different policies in Section 5 using trace data from a data intensive website and two email logs. We present two adaptive policies in Section 6, and conclude in Section 7.

2. RELATED WORK

There has been a considerable amount of research in both pull-based and push-based freshness policies. A widely used pull-based policy is to assign each object a Time-to-Live (TTL) [5, 12], and validate any cached object whose TTL has expired. More recently, work in [17] aims to improve upon this by estimating TTL values based on the probability that an object will be updated within a time interval. Several works, including [11, 6, 17] have suggested modeling updates as a Poisson model. Work in [6, 17] assumes a model that is homogeneous over time, while our proposed model assumes a time varying update intensity, which was shown to work better in [11].

Research reported in [9] considers pre-validation policies to validate cached objects whose TTLs have expired before clients request them, which can reduce the client-perceived latency caused by unnecessary validations (freshness misses).

These policies can reduce the number of validations in response to client requests, however, they *increase* the total number of contacts with the server because objects must be validated offline. In contrast, the policies we present in Sections 5 and 6 can reduce the number of unnecessary validations *and* total contacts with servers for cyclic objects while providing comparable recency to TTL.

Pull-based freshness has also been addressed in the context of synchronizing a large collection of objects, e.g., improving the performance of web crawlers [4, 6, 8]. Updates are detected by periodically prefetching objects from remote sources to maximize the freshness of cached objects. These pull-based policies are not based on complete update histories and therefore may be less accurate.

There has also been much work in push-based freshness [10, 18, 23]. Work reported in [18, 23] shows that push-based freshness is feasible and works well in many cases. However, servers must store information about clients, which may not scale well. Work in [10] proposes an adaptive push-pull scheme where servers can adaptively push updates to some clients and require others to use a pull-based policy. Server driven freshness in the context of caching dynamic web content is considered in [3, 16].

There is also work in push-based freshness that allows cached objects to deviate from objects at the remote server [1, 14, 19, 20]. Servers need to store the inventory of objects in a client's cache and the client's tolerance towards deviation of object values in the cache from that stored on the server side. This may not scale well to a large number of clients.

Piggybacking to improve cache freshness was proposed in [15]. In this work, clients piggyback a list of potentially stale cached objects when they contact a server. Servers piggyback the subset of those objects that have been updated on their responses. This work is orthogonal to ours and is not concerned with estimating the freshness of cached objects, but rather how to efficiently refresh cached objects.

3. MODELING OF UPDATES

In this section, we provide a classification of update patterns and discuss how accurately we can model them, based on an object's update history.

3.1 Update Patterns: An Overview

In general, objects can be classified by the regularity and predictability of their update patterns. At one extreme are objects updated at regular times; at the other extreme are objects with completely unpredictable updates. In this section we present examples of real objects between these extremes. We consider more predictable (cyclic) objects that are updated at similar times each day, but not necessarily at the exact same time each day, so they are not completely predictable. We also consider less predictable (bursty) objects that experience periods with a large number of updates that are not consistent with earlier update patterns. While updates to these objects are not completely random, the bursts are difficult to predict. We analyzed data from the 1998 World Cup website [2] as well as two email logs. We report on the details of these datasets in Section 5.

3.1.1 World Cup

Our analysis of the World Cup data shows that many objects exhibited either cyclic or bursty update patterns. We discuss how we classified objects as cyclic or bursty in Section 6.2. Note that this classification is only for the purpose of reporting our results; our adaptive policies do not require any a priori classification of cyclic or bursty patterns.

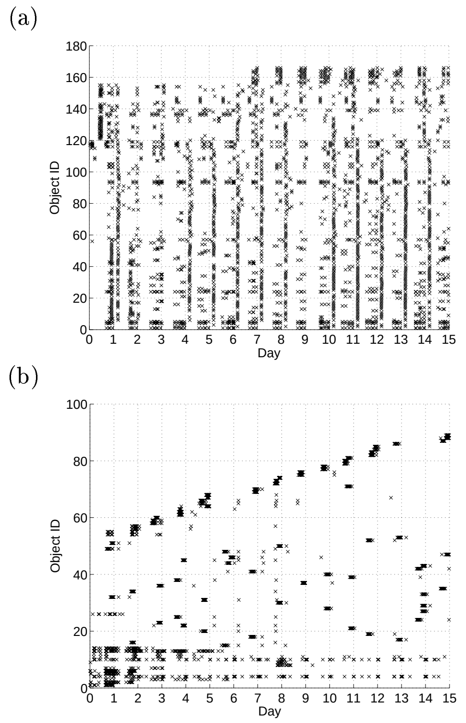


Figure 1: Updates to (a) Cyclic and (b) bursty objects in the World Cup trace

Figure 1 plots the updates to cyclic and bursty objects in the World Cup trace that were updated at least 10 times in a 15-day trace period. In these figures, the x-axis is the time of day within a 15 day window, and each value on the y-axis represents a distinct object. An $\times$ in the graph at point `(x,y)` denotes an update to object `y` at time `x`.

Figure 1(a) shows objects that exhibit cyclic behavior that is repeated daily. For example, we observe in Figure 1(a) that many of the objects are updated at the beginning of each day, although not necessarily at the same time. These objects may correspond to pages that provided daily updates on World Cup scores and events. Cyclic update patterns commonly occur at websites, for example a weather site that updates the temperature at regular times every day.

Figure 1(b) shows objects with bursts of updates. In this trace, these are objects where most of the updates occurred on the same day, and few updates occurred before or after the burst. These objects may correspond to a specific World Cup event such as the score of a match. Many updates to the object occur on the day of the match, but few updates occur on other days. Bursty updates also occur at other web sites, such as news web site that frequently updates an article on the day of a breaking news event.

3.1.2 *Email Traces*

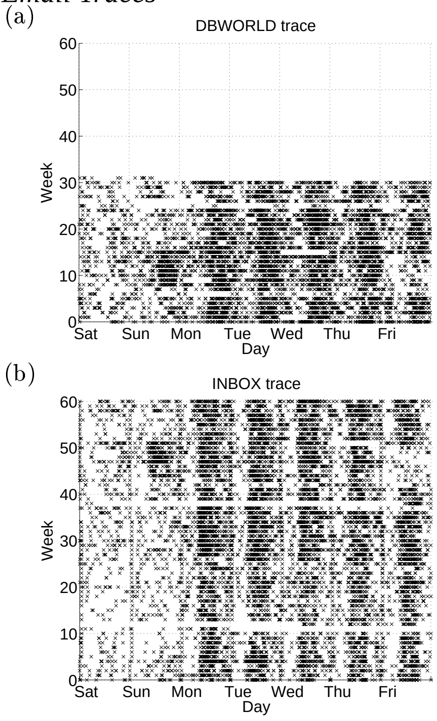


Figure 2: Updates to two email traces

We also considered two different email traces (labeled DBWORLD and INBOX). Figure 2 plots the arrival of email messages in these two traces. In each graph, the x axis shows the day of the week (and relative time of day), and each value on the y axis is a distinct week of the trace. An × value at point (x,y) indicates that an email message arrived in the client's mailbox at time x during week y. The first observation is that both mailboxes exhibit fairly regular behavior from week to week, again showing cyclic update patterns. However, both traces also exhibit occasional bursts of updates (for example, on Sunday around week 50 of the INBOX trace).

3.1.3 *Updates to Cyclic Objects*

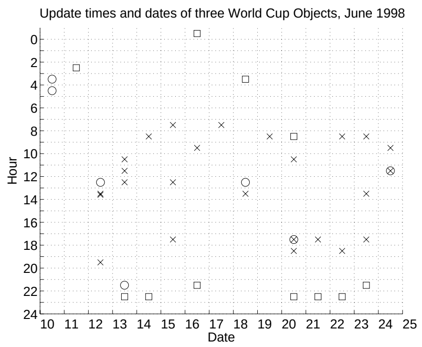


Figure 3: Updates to three distinct cyclic objects in World Cup trace

Figure 3 shows the times of updates to three distinct cyclic objects in the World Cup trace, denoted by three different shapes. The important observation is that they are cyclic

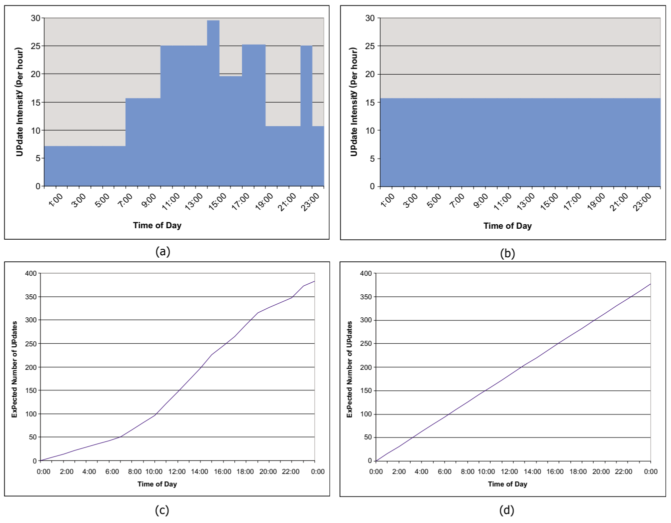


Figure 4: Homogeneous vs. nonhomogeneous update patterns

because they are updated with similar intensity each day at approximately the same time. However, they are not updated at exactly the same time every day, so history information may not accurately predict updates. For example, the object denoted by a × is updated three times on June 13 and 15, but at different times on each of these days. Further, an object may need a sufficiently long history in order to accurately predict updates. For example, the object denoted by a ◯ is updated once between 3:00-4:00 and once between 4:00 and 5:00 on June 10. This history alone would not predict the update that occurs between 12:00-13:00 on June 12. Thus, the effectiveness of update history may be limited by both the amount of history information available and the predictability of updates.

To summarize, objects may exhibit more regular or less regular update patterns. Objects may experience bursts, where a larger number of updates than expected occurs in a given time window. Further, even objects that do not experience bursts may require long individual histories to accurately model updates.

3.2 Modeling Update Patterns

The modeling of update patterns is based on *recurrent piecewise constant update intensities*, as suggested in [11]. The underlying assumption of such models is that there is a time period, e.g., a day, whose update pattern is repetitive. Therefore, one can partition an update history into equal time periods with similar update pattern. To represent update patterns we use a time-varying parameter $\lambda(t)$, representing the intensity of updates over time.

A basic model of update patterns that assumes a *homogeneous* update intensity (λ) over time is inadequate for many applications. Therefore, we present a more refined analysis of λ. Within a given repetitive time cycle, λ may vary, representing, for example, change of intensities between work hours and after hours. Therefore, λ becomes time-dependent. To simplify calculations, one may assume that while λ changes over time, it may be represented as a combination of intervals, in which λ is constant, hence the term *piecewise constant.* To demonstrate the differences between homogeneous and time-dependent λ, consider Figure 4. Figure 4(a) shows the changes to the intensity of updates over a period of one day, using a piecewise-constant model. Figure 4(b) corresponds to a constant arrival rate of updates. Figure 4(c) and Figure 4(d) demonstrate the accumulation of λ (representing, in the case of a Poisson model, the expected number of updates in the corresponding time period) over a period of one day for the time-dependent and homogeneous λ, respectively. While the accumulation for the homogeneous model is linear over time, the accumulation rate of the time-dependent λ changes with fluctuations in the update intensity $\lambda(t)$.

Formally, given a time interval Q, suppose that the update rate $\lambda(t)$ repeats every Q time units, that is, $\lambda(t) = \lambda(t+Q)$ for all t. Furthermore, the interval $[0, Q)$ is partitioned into a finite number of subsets $J_1, \ldots, J_K$, with $\lambda(t)$ constant throughout each J_k, $k = 1, \ldots, K$. Finally, each J_k is in turn composed of a finite number of half-open intervals of the form $[s, f)$. For instance, in Figure 4(a) $k = 7$, with $J_1 = [0{:}00, 7{:}00)$, $J_2 = [7{:}00, 10{:}00)$, etc.

We next define a specific recurrent piecewise constant model. The model is stochastic since the repetitive nature of updates in a distributed autonomous environment cannot be modeled in a deterministic fashion. We use a nonhomogeneous Poisson process [21, 22] with instantaneous update rate $\lambda : \Re \rightarrow [0, \infty)$ to model the occurrence of *update events.* Each *update event* possibly consists of multiple updates (possibly to different data objects) aggregated over an interval in time. The number of update events occurring in any interval $(s, f]$ is a Poisson random variable with expected value $\Lambda(s, f) = \int_s^f \lambda(t)\, dt$. When λ is constant over time $\Lambda = \lambda \cdot (f - s)$. Using the notation given above, each interval J_i will be modeled by a homogeneous Poisson process with its own λ_i.

The expected number of updated objects during $(s, f]$ may be computed as:

$$\int_s^f \lambda(t)dt = \Lambda(s, f) \qquad (1)$$

3.3 Individual and Aggregate History

One may model either individual object history or aggregate history (over a set of objects with similar update patterns). The use of individual history assists in forecasting future updates more accurately, but is costly. A server must maintain the individual history for a potentially large number of objects, as well as indices to rapidly access the history. In addition, accumulating sufficient training data for modeling individual object history may take much longer than the time required for modeling aggregate history. Aggregate history is a less costly alternative in which aggregated data of the *update patterns* of *multiple objects* with common update patterns at a site is used to obtain a model of aggregate update patterns that is sent to the client.

3.3.1 Aggregate History

Time	λ per hour
0-7	23.81
7-10	52.07
10-14,22-23	83.40
14-15	98.53
15-17	65.23
17-19	84.27
19-22,23-24	35.40

Table 1: Aggregate Update History ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) for the World Cup trace

We illustrate how to model aggregate history using the World Cup trace data. We constructed a nonhomogeneous Poisson process to model the update pattern aggregated over all cyclic objects in a training set of eight days of data (from June 10, 1998 to June 17, 1998). 10,074 update times of 4,405 objects were analyzed. While this is a relatively low number of objects, these objects were requested by many clients, and pushing updates to all these clients could be very expensive. We note that the log does not explicitly indicate the time an object is updated. In Appendix A, we describe how we detect updates.

We used a "bulk update" to aggregate updates within 30 seconds of each other into a single event. To simplify presentation, we ignore bulk updates in this work. Assuming a cyclic behavior that repeats daily, we have identified seven distinct segments. Table 1 provides the aggregate history ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$), a vector representing the effective λ value for each time interval. To interpret this history, each row gives the expected number of events per hour during the time interval. For example, between 0:00 and 7:00, there are 23.81 expected updates every hour. To estimate the number of updates to an individual object O in each interval, we scale the λ value by f_o, the fraction of *all updates* at the server that occurred to object O.

It is important to note that in general, models are an idealized representation of a process. It is well known that Poisson processes model a world where updates are independent from one another. Therefore, models such as the one presented above need to be verified. Using verification methods, as suggested in [13], it becomes clear that the World Cup data cannot be accurately modeled using a Poisson model, most likely due to correlations of update events. However, as we show in Section 5, even an "inaccurate" model that considers aggregation over multiple objects can provide a benefit over using only the last modified times of an individual object, and performs on average almost as good as using individual object's history with less overhead.

3.3.2 Individual History

We construct individual history ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$) in the same manner as we construct aggregate history. However, the relatively small number of updates per object makes any segment analysis error prone. For individual object history we partition the day into 24 equal size intervals, and assume a constant λ within each one hour interval. As an example, we consider updates to a single object in the World Cup trace over the 8 day period from June 10- June 17. During this 8 day period, the object had 4 updates in the time period [10:00, 11:00) (which corresponds to λ=0.5, i.e., 0.5 updates/day), 1 update in the time period [11:00, 12:00) (λ=0.125), 1 update in the time period [12:00, 13:00) (λ=0.125), 3 updates in [13:00, 14:00) (λ=0.375), 2 updates in [14:00, 15:00) (λ=0.25). No updates occurred between [17:00, 22:00). Thus, this object experienced a period of high update activity in the morning, moderate activity around noon, another period of high activity in the afternoon, and no activity in the evening.

4. PULL-BASED FRESHNESS POLICIES

We describe three policies that can be used by clients or servers to estimate when objects will be updated, depending on the available history. We note that the focus of this paper is on the policies, and we do not consider the details of where the computation is performed. In general, update history information can be used by either caches or servers to estimate the expiration times of objects, and there is a tradeoff in terms of both flexibility and computational overhead. If clients or caches perform the computation locally, it reduces the load on the server and gives clients greater flexibility in tuning the freshness of their data. However, if the computation is performed at a server or intermediate site, e.g., a proxy, this can reduce computational overhead for clients; this is important when the clients are mobile devices with limited battery power.

4.1 Time-to-Live (TTL)

Using only the time an object was last modified, clients or caches can use TTL, a pull-based policy widely used in practice. TTL estimates how long an object remains fresh in the cache as a function of its *last modification time*. Any object that is estimated to be stale must be validated. TTL can be tuned using a parameter α, which is typically a real number between 0 and 1. If an object is cached at time t_{cache} and was last modified at time $t_{lastmod}$, its TTL is estimated as:

$$TTL = t_{cache} + \alpha * (t_{cache} - t_{lastmod})$$

The TTL policy works as follows: *If a cached object is requested before the TTL time expires, it is served from the cache without validation (i.e., contact with the remote server). If the object is requested after the TTL time expires, the cache validates the object at the remote server before delivering the object.*

Note that smaller values of α generate more conservative TTL estimates, which improve data freshness, but increase the number of validations.

4.2 Aggregate History Based Policy (AggHist)

The aggregate history based policy (AggHist) uses the aggregate history ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) that is learned from the past updates to a set of objects. Table 1 gives an example aggregated over all objects in the World Cup trace. Recall that for a given interval, λ denotes the update intensity.

To estimate the update pattern of an individual object from the aggregate update history, servers scale the aggregate λ values by the relative fraction of server updates f_o that occurred to that object. Without loss of generality, assume that time s falls in interval 0 and time f falls in interval n. Therefore, whenever $n > 0$, we can rewrite Equation 2 to be

$$\mathrm{E}[B(s,f)] = \begin{array}{c} (U(T(0)) - s)\lambda(0) + \\ \sum_{i=1}^{n-1} (U(T(i)) - L(T(i)))\lambda(i) + \\ (f - L(T(n)))\lambda(n) \end{array} \qquad (2)$$

where $U(T(i))$ and $L(T(i))$ represent the upper bound and lower bound of $T(i)$, respectively.

The AggHist policy works as follows: *Given an initial time t_m, an aggregated update history $\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$, the fraction of updates of an object O with respect to the total number of updates at the server f_o calculate the expected number of updates. If the expected number of updates exceeds a threshold θ, then validate the object at the server.*

We illustrate with an example from the World Cup trace. Suppose an object O is cached at 1:00 and requested at 8:00. We use the λ values from Table 1. If 1% of all updates at the World Cup site occur to object O, i.e. f_o= 0.01, the corresponding λ values for the intervals from 1:00 to 8:00 in Table 1 are scaled as follows:

Time [1:00 - 7:00]: 23.81 * 0.01 = 0.2381

Time [7:00 - 8:00]: 52.07 * 0.01 = 0.5207

The expected number of updates is:

$ExpUpd(1,7) + ExpUpd(7,8)$ = (0.2381 * 6 hours) + (0.5207 * 1 hour) = 1.95

4.3 Individual History Based Policy (IndHist)

The individual history policy (IndHist) uses the individual history ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$) to estimate the freshness of a cached object.

The IndHist policy works as follows: *First, use $\vec{H}_{ind}$ to estimate the expected number of updates to the cached object. Use formula 2 to compute the expected number of updates. If the expected number of updates exceeds a threshold θ, validate the object.*

Using our example object from Section 3.3.2, if the object was cached at 11:30 and is requested at 14:00, its expected number of updates is $\frac{1}{2}ExpUpd(11,12) + ExpUpd(12,13) + ExpUpd(13,14) = \frac{1}{2} * 0.125 + 0.125 + 0.375 = 0.5625$.

5. COMPARISON OF TTL, INDHIST, AND AGGHIST

We now evaluate the AggHist, IndHist, and TTL policies on data traces that exhibit both cyclic and bursty behavior. We first compare TTL and IndHist on the two email traces, which exhibited cyclic update patterns. We then compare TTL, AggHist, and IndHist on both cyclic and bursty objects in the World Cup trace, and motivate the need for a new class of adaptive policies that can choose among policies according to an object's update patterns.

5.1 Data Traces

5.1.1 *World Cup Data*

The trace data from the 1998 World Cup Web Site [2] contains a log of all requests to the site. The World Cup site had servers in four different geographical locations: Paris, France; Herndon, VA; Santa Clara, CA; and Plano, TX. The entire trace consists of 1.3 billion requests made from May 1, 1998 to July 23, 1998. In our experiments we used a 15-day subset of this trace from June 10, 1998 to June 25, 1998. This corresponds to the first 15 days of the World Cup event and includes about 333 million requests. In our experiments, we report separate results for cyclic and bursty objects, although our policies do not require this classification. To identify objects in each category, we classified objects offline using the update histories from all 15 days of the trace, using the techniques to be described in Section 6.2.

For each request, the trace contains the following:

- *ClientID:* Unique ID of the client making the request. Note that this may be a proxy.
- *ObjectID:* Unique ID of the requested object.
- *Timestamp:* The time the request was made.
- *Size:* Size of the object in bytes.

The trace does not explicitly give information on updates to objects, however, we can infer updates when an object changes size as described in Appendix A.

In the 15-day trace, 42 million requests were for cyclic objects and 11 million requests were for bursty objects. If all clients had sufficient cache space (see Section 5.2), 9 million of the requests for cyclic objects would be cache hits (but not necessarily up to date in the cache), as would be 1 million of the requests for bursty objects. Note that the percentage of requested bursty objects that are in the cache ($\approx$11%) is smaller than for cyclic objects ($\approx$21%). This is because the bursty objects are most interesting to clients during a short interval (during the bursty period), so they are less likely to be cached prior to the update burst.

5.1.2 Email Data

Our first email trace (**DBWORLD**) includes email notifications of postings to the DBWORLD electronic bulletin board and other messages. The data were collected over seven months and consists of more than 6400 insertions, from November 9, 2000 through June 17, 2001. Our second email trace (**INBOX**) is taken from messages to a client's inbox from March 3, 2001 - May 24, 2002 and consists of about 10,000 insertions. We collected the data for both these traces using a capture program (similar to the way the vacation program works on Unix) to capture messages and process them.

5.2 Setup

5.2.1 World Cup Experiments

Our experiments with the World Cup trace model a traditional web caching scenario. When a client requests a cached object, the cache uses the policy to determine whether or not to validate the object. Using TTL, an object is validated if it is requested after it expires. Using IndHist and AggHist, it is validated if the expected number of updates exceeds a specified threshold θ.

We maintained separate caches for each client ID, which may correspond to either an individual client or a proxy. For each client ID, we assumed an initially empty cache. To simplify our presentation, we assume all clients had sufficient space to cache their objects and no objects were evicted from client caches during the trace period. This is a reasonable model because cache size affects only the hit rate of the cache. Therefore, a limited cache would have equal impact on the performance of all estimation policies, and would not change their relative accuracy. Each experiment included a training period to gather object update history information, followed by a test period during which we collected data. We give the length of the training and test periods when reporting the results of each experiment.

Most bursty objects in the World Cup trace had a “burst” of updates on a single day, and few (if any) updates on other days, as shown in Figure 1(b). For these objects (or any object with no history available), TTL is likely to provide more accurate freshness estimates compared to the IndHist based policy. A more interesting case occurs when an object that normally has cyclic update patterns experiences a burst in updates. This could occur at a news web site that is normally updated at regular intervals but experiences a burst of updates during a breaking news event. For these objects, IndHist is likely to do well during cyclic periods, but TTL may do better during a burst.

Few objects in the World Cup trace exhibited this behavior of cyclic patterns and bursts. We modified the trace data as follows to generate such objects. We randomly selected 55 of the most popular bursty objects with respect to client requests and mapped them to 55 of the most popular cyclic objects. In our experiments on bursty World Cup objects, we treated each bursty/cyclic pair as a single object. These 55 merged objects exhibited cyclic update patterns for most of the 8 days, but experienced bursts of updates on one day. These were used for the experiments in Sections 5.3.2.2 and 6.2.

5.2.2 Email Experiments

Our experiments with the email traces model a scenario where a client has a locally cached mailbox, e.g., on their mobile device, that needs to be refreshed in the background to promptly notify the client of new messages. The goal is to minimize the time elapsed between when a new message arrives and when it appears in the client's mailbox. This differs from the above web caching application where objects are refreshed only when they are requested (and the cached copy is not sufficiently fresh). We note that web caching with prefetching to refresh cached objects is similar to the email case.

For the email application, we compare the TTL and IndHist policies. After each refresh, for the TTL policy, we computed the time of the next refresh as a function of the time the last message arrived. For the IndHist policy, after each refresh we computed the time of the next refresh as the time that the expected number of updates (i.e., new messages) would exceed some threshold θ. We used the first week of each trace as a training period to gather a history, and continuously updated the history during the experiments.

5.2.3 *Metrics*

We use the following metrics:

- **Total Validations**: This is the number of times requested objects that were in the cache needed to be validated at the remote server.
- **Stale Hits**: For the World Cup trace, this is the number of objects that were served from the cache without validation but had actually been updated at the remote server.
- **Average Delay**: For the email traces, this is the average amount of time elapsed between the arrival of a new message and the time it appears in the client's mailbox.

5.3 Results

Our experiments show that using either IndHist or AggHist for cyclic objects can *significantly improve the accuracy of estimates of an object's freshness*. In web caching, this can increase the number of objects served from the cache without validation, which reduces costly remote server accesses for clients and reduces unnecessary contacts with servers, which can be as high as 30-50% of all cache hits [9]. In email applications, this can reduce the delay of new messages appearing in a client's mailbox without increasing the mailbox refresh rate, which is of particular importance to mobile devices.

5.3.1 *Accuracy of Estimates*

5.3.1.1 *World Cup Trace*

We first compare the accuracy of estimating the number of updates to cyclic objects in the World Cup Trace using TTL, IndHist, and AggHist. Each time a client requests a cached object, we compare the actual number of updates to the object against the estimated number using each policy. Using TTL, we estimate the number of updates to an object at time t as $(t - t_{lastmod})/(TTL - t_{lastmod})$, where $t_{lastmod}$ is the last modified time of the object, and use an α value of 0.05, which is typical of values used in practice. We note that other values of α do not show significantly different results. For IndHist and AggHist, we calculate the estimated number of updates to an object as described in Section 4.

Figure 5 compares the estimated updates to the actual value for each policy. A value of 0 means the estimate was *accurate*. A positive error value means the actual value *exceeded* the estimated value, and a negative value means the actual value was *less* than estimated. AggHist and IndHist have nearly *twice* as many accurate estimates as TTL. This shows that using histories can significantly improve the accuracy of freshness estimates for cyclic objects.

5.3.1.2 *Email Traces*

We next consider the accuracy of the IndHist policy for the email application. We do not consider AggHist for this application because the trace consists of a single object (the mailbox). Recall that for the email application, using IndHist we refreshed the mailbox whenever the expected number of updates (new messages) exceeded θ. In Table 2 we compare the expected number of these updates per validation against the actual number of updates per validation. We report results for the DBWORLD trace (results for the INBOX trace were comparable). The expected number and actual number of updates are very close and shows that IndHist accurately estimates the number of updates.

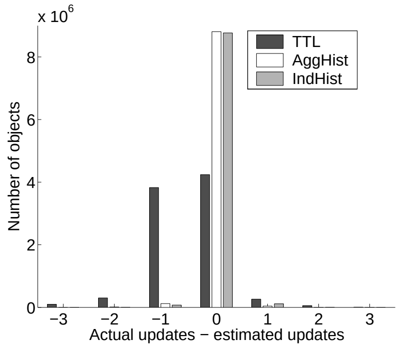


Figure 5: Comparison of three policies for Cyclic objects in the World Cup Trace

expected	actual
0.1	0.098
0.2	0.196
0.5	0.493
0.7	0.691
1.0	0.980

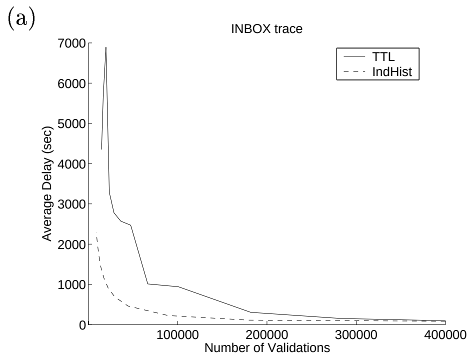
Table 2: Comparison of the expected and actual number of updates using IndHist

5.3.2 *Number of Validations*

We now report on the number of validations required to maintain a given level of freshness.

5.3.2.1 *Email Traces*

We first consider the DBWORLD and INBOX traces. Recall that we use the average delay as our metric. We tune TTL by varying α between 0 and 1 and IndHist by varying θ between 0 and 1. We plot the number of validations against the average delay for TTL (for different α values) and IndHist (for different θ values) in Figure 6. As expected, as the number of validations increases, the average delay decreases. The key observation is that for a given average delay, IndHist performs significantly fewer validations than TTL. For example, in Figure 6(a), to provide an average delay of about 500 seconds, TTL must perform about 170,000 refreshes while IndHist performs about 80,000, i.e., a 47% reduction in the number of validations. Similarly, in Figure 6(b), to provide an average delay of 500 seconds TTL performs about 50,000 validations while IndHist performs about 20,000, i.e., a 60% reduction. Thus, IndHist can reduce the total number of refreshes by more than half. This can provide significant savings in terms of both power and bandwidth to clients who read email on their mobile devices. Recall that the email traces generally exhibited



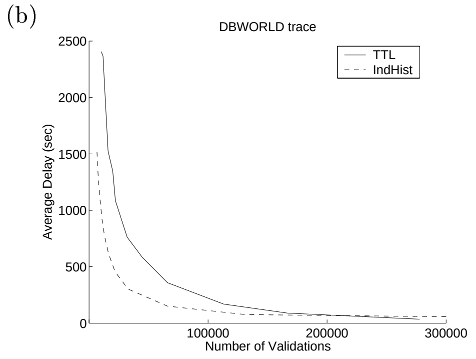


Figure 6: Effect of Tuning TTL and IndHist on average delay and validations

cyclic behavior. These results show that IndHist can indeed perform better than TTL for cyclic objects.

5.3.2.2 *World Cup Trace*

Next, we compare the TTL, IndHist, and AggHist in terms of both number of validations and data freshness of both cyclic and bursty objects in the World Cup trace. In the experiments for cyclic objects, we used all 15 days of trace data. We used the first 8 days to construct histories, and ran the experiments on the next 7 days. In the experiments for bursty objects, we used 8 days of trace data and performed preprocessing as described in Section 5.2.1. We used the first 4 days to construct individual update histories, and ran the experiments on the last 4 days. For all three policies, we varied the tuning parameter from 0.05 to 0.7.

In Figure 7 we report on the number of stale hits given similar levels of total validations. Note that for all values of θ, IndHist did not go beyond 2,500,000 validations and AggHist did not go beyond 3,500,000 validations. For all three policies, increasing the total number of validations reduces the number of stale hits. Given the same number of validations, both AggHist and IndHist deliver significantly fewer stale objects than TTL. This is because the improved accuracy of the freshness estimates of objects reduces the number of unnecessary validations, and shows once again that using histories can perform better than TTL for cyclic objects. This is especially true when there are relatively few validations, i.e., higher values of α and θ. For example, when each of the policies has about 1,500,000 total validations, TTL (α $\approx$0.5) provides $\approx$800,000 stale hits while AggHist

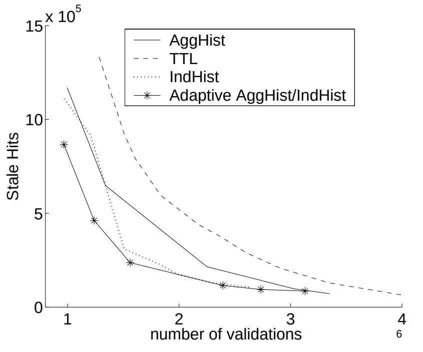


Figure 7: Effect of Tuning TTL, AggHist, and IndHist on data freshness and validations for cyclic World Cup objects

provides $\approx$500,000 stale hits and IndHist provides $\approx$300,000 stale hits.

A key observation is that IndHist offers an overall improvement over AggHist because it can model the individual update patterns of objects that may differ from the average behavior. However, as shown in Figure 3, individual history is not always a good predictor of updates for cyclic objects. If there is insufficient history information available, the IndHist policy may not be able to accurately predict when updates will occur. In contrast, since AggHist captures the behavior of objects with similar update patterns, it is better suited to deal with new objects whose history is too short to yet be stable. Figure 7 compares an adaptive IndHist/AggHist policy (described in detail in Section 6) that can combine the benefits of both policies. This adaptive policy *performs better than either policy alone.* The adaptive IndHist/AggHist policy chooses between IndHist and AggHist based on the available history information, as described in the next section.

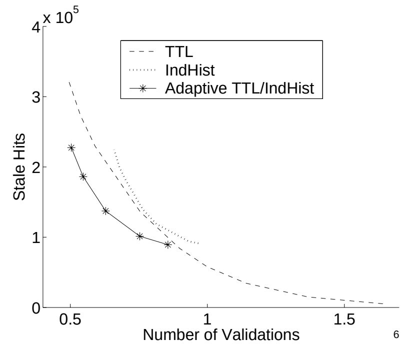


Figure 8: Effect of Tuning TTL and IndHist on data freshness and validations for bursty World Cup objects

Figure 8 compares TTL and IndHist on the bursty objects in the World Cup trace. AggHist performed significantly worse than both of these and we do not show these results. As expected, TTL performs better than IndHist (yet not significantly better) because it can more accurately predict when updates occur during bursty periods. However, we expect that IndHist is more effective in non-bursty periods. To illustrate, an adaptive TTL/IndHist policy is shown in Figure 8; it can perform better than TTL alone because it uses IndHist to estimate updates during non-bursty periods. In Section 6 we evaluate this adaptive TTL/IndHist policy on both cyclic and bursty objects.

6. ADAPTIVE PULL-BASED POLICIES

Figures 7 and 8 illustrate that adaptive policies that can choose among multiple techniques outperform a less flexible policy limited to one technique. There are several challenges in developing an effective adaptive policy. One is choosing the individual techniques. The second is choosing the criteria to choose techniques, and the third is making sure the choice is beneficial. We present two adaptive policies, Adaptive IndHist/AggHist and Adaptive IndHist/TTL. We note that there are many other adaptive policies that could be used, and we do not claim that these two are the best. Rather, our contribution is in motivating the use of adaptive policies, and illustrating their use and benefits.

6.1 Adaptive IndHist/AggHist

We have observed that for cyclic objects, IndHist works well when there is sufficient individual history available to predict when the object is likely to be updated. In contrast, AggHist works well for objects with cyclic behavior, but whose histories are too short to be stable. An adaptive policy can exploit both the aggregate and individual behavior for cyclic objects.

6.1.1 Policy

We present a criteria for adaptive IndHist/AggHist to choose between IndHist and AggHist. Note that there may be many such criteria.

Suppose *NumHours* is the number of distinct hourly intervals when the object had an update. Intuitively, if the total number of updates to the object *NumUpdates* is equal to *NumHours*, i.e., the ratio $NumHours/NumUpdates = 1$, then the object was updated at different times each day. This suggests that there is insufficient history available to accurately model updates using IndHist, and AggHist may give a better approximation of the object's behavior. On the other hand, if the object was updated repeatedly in the same (hourly) interval, then $NumHours/NumUpdates < 1$. This suggests that there is sufficient history available to model the object accurately using IndHist.

The Adaptive IndHist/AggHist policy is as follows: *Given an individual history and a parameter $T \leq 1$, we compute the ratio NumHours/NumUpdates. If NumHours/NumUpdates $> T$, AggHist is used; else IndHist is used.*

Note that for T=1, IndHist is always selected, and for T=0 AggHist is selected. Clients can more aggressively choose AggHist or IndHist by varying the value of T; this is discussed in the experiments.

6.1.2 Results

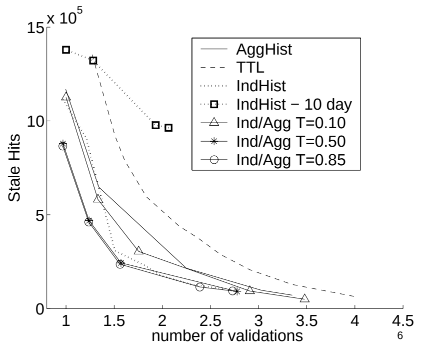


Figure 9: Comparison of Adaptive IndHist/AggHist to TTL, IndHist, and AggHist

We report on Adaptive IndHist/AggHist for different T values for cyclic objects in the World Cup Trace.

Figure 9 plots the number of validations compared to the number of stale hits. Each curve is a different value of T. We first consider the impact of using a shorter update history or limited update information on the accuracy of the IndHist policy. We limited the length of the history to 10 days prior to the time the object was cached (labeled IndHist- 10 day in Figure 9). Note that in contrast, the policy labeled IndHist includes all prior updates to an object in the trace, up to all 15 days. For many objects, the IndHist-10 day policy does *worse* than TTL and AggHist. This motivates the need for sufficient history for IndHist, and shows the benefits of AggHist when there is insufficient history available for an individual object.

Next, we compare the performance of the Adaptive IndHist/AggHist policy to IndHist (with complete update histories) and AggHist alone. Adaptive IndHist/AggHist with T ranging from 0.50 to 0.85 outperforms TTL, IndHist, and AggHist. When T= 0.1, the policy performs closer to AggHist as expected. These results suggest that Adaptive IndHist/AggHist is not very sensitive to the selection of T and any value around 0.5 that allows the adaptive policy to switch is effective.

6.2 Adaptive IndHist/TTL

For bursty objects, the IndHist policy performs best during the non-bursty periods, but performs poorly when objects experience bursts. Further, when it uses a full update history, including bursty periods, it may estimate updates less accurately after the bursty period. In contrast, TTL performs best during bursty periods because it assumes that objects that were recently updated are likely to be updated again soon. An adaptive IndHist/TTL policy presented in Figure 8 combines the best features of both policies.

Our adaptive IndHist/TTL policy can detect bursts and dynamically chooses between IndHist and TTL. Thus, it can generalize well to different types of update patterns, and requires no prior knowledge of whether an object is cyclic or bursty. We first describe how we identify bursts. We then describe the adaptive policy (Adaptive IndHist/TTL) which dynamically chooses between the IndHist and TTL policies depending on whether or not an object exhibits bursty behavior. Thus, for objects with no bursts, it has comparable performance to IndHist.

6.2.1 Identifying Bursts

We use the term *burst* to refer to the case where the number of actual updates to an object is considerably higher than that approximated by IndHist. Consider an object that is cached at time t and created at time t_0. We estimate that a burst occurs when the *actual number of updates* in a window of size W prior to t, i.e., all updates in the interval $[t-W,\ t]$, exceeds the *expected number of updates.* The *expected number of updates* is estimated by IndHist, using only updates that occurred in $[t_0,\ t-W]$ prior to the current cycle.

The adaptive history policy works as follows: *Given an intensity function λ for the interval $(t_0,\ t$ - $W)$, and λ^* for the interval $(t-W,\ t)$, a distance function $f(\lambda, \lambda^*)$, and a threshold T, Adaptive IndHist/TTL identifies a burst if $f(\lambda, \lambda^*) \geq T$. On each request, if $f(\lambda, \lambda^*) \geq T$, Adaptive IndHist/TTL assumes a burst is occuring and uses TTL. Else, if $f(\lambda, \lambda^*) < T$, Adaptive IndHist/TTL assumes a burst is not occuring and uses the IndHist policy.*

We next provide a distance measure f. This was empirically evaluated to provide a good estimation of bursty periods in the World Cup trace data. We note that more research is needed to identify distance measures that will work on several traces. Let the *expected number of updates* (Λ) in $[t-W,\ t]$ with respect to time t be $\Lambda(W,t)$, and the *actual number of updates* (Λ^*) in $[t-W,\ t]$ be $\Lambda^*(W,t)$. Then,

$$f(\lambda, \lambda^*) = \begin{cases} \frac{\Lambda^*(W,t)}{\Lambda(W,t)} & \text{if } \Lambda(W,t) > 0 \quad (a) \\ T & \text{if } \Lambda(W,t) = 0 \text{ and } \Lambda^*(W,t) > 0 \quad (b) \\ 0 & \text{otherwise} \quad (c) \end{cases}$$

Intuitively, condition (a) covers the case when at least one update was expected $(\Lambda(W,t) > 0)$. A burst occurs when the ratio of *observed* updates to *expected* updates exceeds T. Condition (b) covers the case when no updates were expected $(\Lambda(W,t) = 0)$ and at least one update occurs. Note that when T=0, this policy is identical to TTL, and when T=∞, this policy is identical to IndHist, thus, it is a generalization of these two policies.

6.2.2 Policies

We compare TTL, IndHist, and Adaptive IndHist/TTL. We evaluate the policies on both the "combined" trace of 55 merged objects and on the remaining cyclic objects. We ran these experiments on the first 8 days of our 15 days of trace data. We used the first 4 days to gather history information, and report results on the remaining 4 days. For comparison purposes, we also report on results for the cyclic objects during the same period.

For Adaptive IndHist/TTL, recall that we estimate when a burst occurred by considering the number of updates in a window W. Adaptive IndHist/TTL will use TTL whenever $f(\lambda, \lambda^*)$ in a window of size W exceeds the threshold T. In our experiments, we report results for W = 1 hour and W = 24 hours, and T= 2.

6.2.3 Results

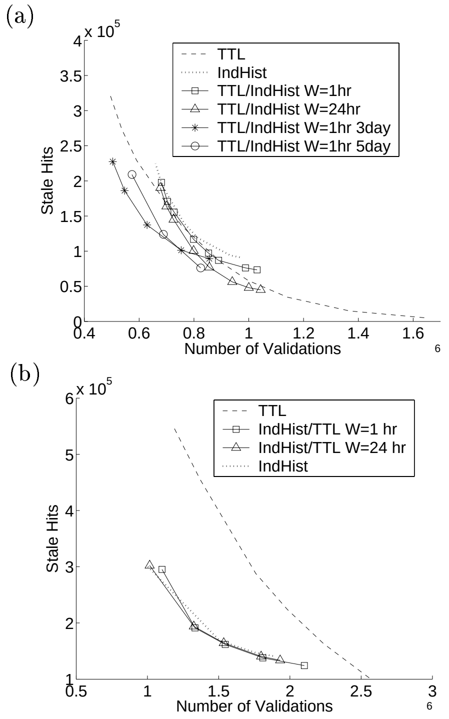


Figure 10: Effect of Tuning TTL, IndHist, and Adaptive IndHist/TTL on data freshness for (a) bursty and (b) cyclic objects

We compare the performance of TTL, IndHist, and Adaptive IndHist/TTL. For all policies, we varied the tuning parameter from 0.02 to 0.7. We plot the number of stale hits versus the total number of validations in Figure 10(a). As expected, TTL outperforms IndHist for the bursty objects. This is because TTL assumes that objects that have been updated recently are more likely to be updated soon, so it is well-suited during bursty periods. In contrast, IndHist assumes that an object's update patterns will be consistent with its past update history, so it cannot handle bursts as well. However, Adaptive IndHist/TTL offers improvement over IndHist, especially as the total number of validations increases. This shows that Adaptive IndHist/TTL can detect some bursts in updates and chooses TTL when appropriate. Adaptive IndHist/TTL with W=24 provides fewer stale hits, and in most cases provides fresher data than TTL for the same number of validations. This suggests that larger values of W may be more effective at detecting bursts.

We also consider the effects of truncating update histories for bursty objects. We consider maintaining a sliding window of individual update histories and ignoring updates that occured outside this window. We consider windows of both 3 days and 5 days prior to the time an object was cached. We show these results in Figure 10 (a) for Adaptive IndHist/TTL with W=1hr. The interesting observation is that using shorter update histories makes the adaptive policy perform better than TTL. This contrasts with the results in Figure 9, which showed that shorter update histories caused IndHist to perform worse than TTL for cyclic objects.

We hypothesize that shorter update histories perform well for objects that experience bursts because they allow ignoring the bursty period. This improves the accuracy of the IndHist policy. In contrast, shorter update histories make IndHist perform worse for cyclic objects because they decrease the amount of available history information.

Thus, our results show that (1) Adaptive IndHist/TTL performs better than either TTL or IndHist alone for objects with both cyclic and bursty periods and (2) when using histories to predict updates, it is important to ignore bursty periods that are not consistent with the rest of the history. Ignoring bursty periods (by using shorter histories) significantly improves the accuracy and effectiveness of Adaptive IndHist/TTL. Developing techniques to detect bursts, and to ignore them is an area for future work.

We also compare the performance of IndHist and Adaptive IndHist/TTL on the cyclic objects over the same 8-day period. Our goal is to ensure that Adaptive IndHist/TTL performs as well as IndHist on cyclic objects. We plot these results in Figure 10 (b). The key observation is that Adaptive IndHist/TTL has comparable performance to IndHist for cyclic objects, so it can generalize to both cyclic and bursty objects without requiring any a priori classification of an object's behavior.

7. CONCLUSIONS

The growing popularity of wide area applications requires freshness solutions that can provide fresh data for objects with a wide variety of update patterns in the presence of limited connectivity. In this paper, we have presented a class of adaptive solutions that use object update histories to choose the most appropriate freshness policy for an object. Our solutions require no a priori classification of objects and can automatically adapt to changes in object behavior. Further, since our solutions do not require servers to store any information about clients and their caches, they can scale to a large number of clients. We have evaluated the effectiveness of using update histories and adaptive policies on trace data from two different applications, and shown that both update histories and adaptive policies are effective ways to improve the accuracy of pull-based data freshness policies. In future work, we plan to design a general framework for evaluating the efficiency of various adaptive policies.

8. REFERENCES

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM. TODS Vol. 15, no. 3*, 1990.

[2] M. Arlitt and T. Jin. 1998 world cup web site access logs. *Available at http://www.acm.org/sigcomm/ITA/*, 1998.

[3] K.S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. *Proc. SIGMOD*, 2001.

[4] D. Carney, S. Lee, and S. Zdonik. Scalable application-aware data freshening. *Proc. ICDE*, 2003.

[5] V. Cate. Alex - a global filesystem. *Proc. USENIX File System Workshop*, 1992.

[6] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *Proc. ACM SIGMOD Conf.*, 2000.

[7] J. Cho and H. Garcia-Molina. Estimating frequency of change. *ACM. TOIT*, 3(3), 2003.

[8] J. Cho and A. Ntoulas. Effective change detection using sampling. *Proc. VLDB Conf.*, 2002.

[9] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. *Proceedings of IEEE INFOCOM*, 2001.

[10] P. Deolasee, A. Katkar, P. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *Proc. 10th WWW Conf.*, 2001.

[11] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: a stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.

[12] J. Gwertzman and M. Seltzer. World wide web cache consistency. *Proc. USENIX Technical Conference*, 1996.

[13] R.V. Hogg and E.A. Tanis. *Probability and Statistical Inference.* MacMillan, New York, second edition, 1983.

[14] Y. Huang, R. Sloan, and O. Wolfson. Divergence caching in client-server architectures. *Proc. PDIS*, 1994.

[15] B. Krishnamurthy and C. Wills. Study of piggyback cache validation for proxy caches in the world wide web. *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[16] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. *Proc. VLDB*, 2001.

[17] J.-J. Lee, K.-Y. Whang, B. S. Lee, and J.-W. Chang. An update-risk based approach to ttl estimation in web caching. *Proc. Conference on Web Information Systems Engineering (WISE)*, 2002.

[18] C. Liu and P. Cao. Maintaining strong cache consistency on the world wide web. *Proc. ICDCS*, 1997.

[19] C. Olston, B.T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *Proc. ACM SIGMOD Conference*, 2001.

[20] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. *Proc. ACM SIGMOD Conference*, 2002.

[21] S. Ross. *Stochastic Processes.* Wiley, second edition, 1995.

[22] H.M. Taylor and S. Karlin. *An Introduction to Stochastic Modeling.* Academic Press, 1994.

[23] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering server-driven consistency for large scale dynamic web services. *Proc. 10th WWW Conf.*, 2001.

Appendix A: Preparation of World Cup Data

In the World Cup trace, we detected an update whenever an object's size changed in the trace. However, some changes to an object's size were not due to updates. Many apparent changes in an object's size were caused by temporary inconsistencies at servers in different geographic locations. Our solution to this problem was to only consider an object changed when the majority of requests to the object had the new size, and when the object had this size for at least two minutes. This allowed enough time for updates to propagate to servers in all four locations, to eliminate the effects of false changes due to server inconsistencies.