# Maintaining Temporal Coherency of Virtual Data Warehouses

*Raghav Srinivasan* *Chao Liang* *Krithi Ramamritham*
Dept of Computer Science, University of Massachusetts, Amherst, MA 01002
{raghav,cliang,krithi}@cs.umass.edu

## Abstract

*In Electronic Commerce applications such as stock trading, there is a need to consult sources available on the web for informed decision making. Because information such as stock prices keep changing, the web sources must be queried continually to maintain* temporal coherency *of the collected data, thereby avoiding decisions based on stale information. However, because network infrastructure has failed to keep pace with ever growing web traffic, the frequency of contacting web servers must be kept to a minimum. This paper presents adaptive approaches for the maintenance of temporal coherency of data gathered from web sources. Specifically, it introduces mechanisms to obtain timely updates from web sources, based on the dynamics of the data and the users' need for temporal accuracy, by judiciously combining push and pull technologies and by using* virtual data warehouses *to disseminate data within acceptable tolerance to clients. A virtual warehouse maintains temporal coherence, within the tolerance specified, by tracking the amount of change in the web sources, pulling the data from the sources at opportune times, and pushing them to the clients according to their temporal coherency requirements. The performance of these mechanisms is studied using real stock price traces. One of the attractive features of these mechanisms is that it does not require changes to either the web servers or to the HTTP protocol* [1].

**Keywords**: Electronic Commerce, Temporal Correctness, Data warehouses, World Wide Web, Cache.

## 1. Introduction

Today many applications find it necessary to consult sources available on the web for informed decision making. Given the autonomy of many of these data sources, as well as the temporal nature of some of the data, it may not be possible to materialize the state of the world as represented in these data sources. Hence the data brought together from the various sources can at best be described as a *virtual warehouse (VW)*. The process of gathering data from distributed sources and maintaining their consistency has to be done efficiently and correctly. Applying workflow ideas, we have developed a system that gathers required information without a user explicitly asking for each and every piece of relevant information [KRGL97]. The collection and collation of relevant information is expressed via workflow specifications and the system automatically accesses the necessary sources. Knowing the format of an HTML page, this system has the capability to parse an HTTP reply and extract relevant information, such as stock prices, from the retrieved pages. It then integrates the necessary information using a CGI script and presents it to a user.

A challenging issue here is the maintenance of *temporal coherency*, that is, keeping the VW's deviation from the real world minimal in the temporal dimension. As an example, consider stock trading data. The need for a system that maintains temporal coherency of a VW containing information needed for intelligent decision making by stock investors is obvious. In practice, it is important to keep the data *only as up to date as is needed by the user* of a data item. For example, if a stock broker desires to sell stocks only when the price goes up by one dollar, smaller increases in the stock price will not be relevant, and hence, need not be communicated to him/her. Exploiting such requirements is one way to minimize network and system overheads incurred in the maintenance of temporal coherency. How this can be done is the subject of the paper.

We introduce mechanisms to obtain timely updates from web sources, based on the dynamics of the data and the users' need for temporal accuracy, by judiciously combining push and pull technologies and by using VWs to disseminate data within acceptable tolerance. Specifically, the virtual warehouses (maintained by client organizations) ensure the temporal coherence of data, within the tolerance specified, by tracking the amount of change in the web sources. Based on the changes observed and the tolerance specified by the different clients interested in the data, the VW determines the time for *pull*ing from the server next, and pushes newly acquired data to the clients according to their temporal coherency requirements.

[1]Supported by NSF grant IRI-9619588. Part of this work was done while the third author was at IIT, Mumbai, on leave from the Univ. of Mass. Amherst.

Of course, if the web sources themselves were aware of the clients' temporal coherency requirements and they were endowed with *push* capability, then we can avoid the need for mechanisms such as the ones proposed here. Unfortunately, this can lead to scalability problems and may also introduce the need to make changes to existing web servers (which do not have push capabilities) or to the HTTP protocol. Our mechanisms do not suffer from these disadvantages, making them especially attractive compared to other protocols that have been proposed for maintaining cache consistency in the Web context.

The rest of the paper is organized as follows - Section 2 explains the problem of maintaining temporal coherency, presents the issues involved in developing efficient solutions and discusses prior related work. Section 3 discusses different algorithms for determining *when* data sources must be contacted to obtain up-to-date data values so as to meet user-level temporal coherency requirements. Section 4 analyzes the performance of the algorithms using real-world stock market data streams. In Section 5, summarizing remarks and the directions for future work are presented.

## 2. Maintaining Temporal Coherency

In this Section, we discuss the problem of maintaining cache coherency and the issues involved in solving it. Finally, we discuss prior approaches to dealing with this problem.

### 2.1. The Problem

Consider a (stock trading) company that has many clients (i.e., stock brokers). Each broker focuses on one or more stocks and is interested in all the information needed to make decisions regarding those stocks, specifically, information about many different stocks, their competitors' stocks, company profiles, etc. So it is quite conceivable that information brought to serve the needs of one broker may be useful for another broker, especially if many of them are focusing on the same "hot stock" of the day. Also different brokers may have different temporal consistency requirements for the different stock prices that they are interested in. Under these circumstances, it makes practical sense to build a single VW for the whole company thereby serving the needs of the different clients from this single VW. Here, the state of the real-world objects is maintained in the web servers, and the images are maintained by the VW, and also in the users' (i.e., brokers') views. The VW acts as a *cache* containing the data from various web servers.

*Temporal coherency* is concerned with the relationship between an object in the real world and its image in a database [KRAM93, HZFJ98]. An interesting and challenging problem arises when we recognize that as time progresses, data in the warehouse gets more and more out of synch with the sources. That is, temporal coherency may be lost as time progresses. This is especially true of the more dynamic (i.e., volatile) data such as stock prices.

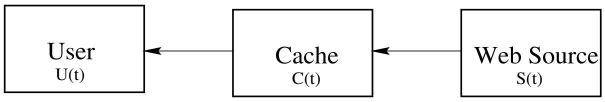


**Figure 1. Real World Objects and their Images**

Consider Figure 1. Here, $S(t)$ denotes the data source value, $C(t)$ and $U(t)$ denote the values at the cache and the user respectively – all at time $t$. In an application such as stock trading, users would rather remain oblivious to minor changes in stock prices. In terms of temporal coherency requirements, every user specifies a constraint, $c$, which means that changes of magnitude less than $c$ in source data need not be informed to the user but the user should be aware of changes whose magnitude is greater than $c$. Thus, the system must guarantee

$$|U(t) - S(t)| < c$$

An example of a user constraint in the value domain is, *Update stock price of Company X whenever the stock price at the source changes by more than 50 cents*. An example of a user constraint in the time domain is, *Update the stock price relating to Company Y every 5 mins*. Either of these constraints can be easily satisfied if the server is aware of them. However, forcing the server to keep track of users' needs and to make it responsible for transmitting the changes to users leads to the scalability and fault-tolerance problems typically associated with stateful servers. Partly due to this, web sources are designed such that users will have to contact a server to obtain the latest state of the information maintained by the server. Thus, the problem of maintaining the user constraint translates to *how often* the source must be contacted. This is obviously straightforward when a user constraint is stated in the time domain. It is not, when constraints in the value domain must be translated into constraints with respect to the time domain, i.e., into requirements on *when* data at a client needs to be updated.

Thus, the issue is one of keeping the *deviation* of temporal coherency within user-specified bounds. We must employ efficient data refreshing schemes by which dynamic information such as stock prices can be provided to the brokers as prices change. This paper introduces mechanisms to achieve timely updates to the virtual warehouse, based on the dynamics of the data and the users' need for temporal accuracy, by judiciously combining push and pull technologies and by using *cache servers* to disseminate data within acceptable tolerance.

### 2.2. The Issues

As portrayed in Figure 1, to users, the cache server acts as a server while the web sources act as servers to the cache server. How to maintain consistency between the source, the cache and the user is the main issue addressed in this paper.

In a typical client-server environment, maintaining consistency of the client and the server can be achieved in one of two ways: 1) In the `Server Push` model, the source of the data, i.e., the server, *push*es data to the client (whenever data changes at the source). In this model, the server can keep track of user requirements and push the data to a client at the appropriate time. Since the cache server maintains the cache and is the server for the clients we can use the server push mechanism to transmit data to the users. 2) In the `Client Pull` model, the client *pulls* data from the source (whenever it suspects that data might have changed at the source). So, consistency of the data at the client depends on how often the client polls the server. Too frequent polling may result in unnecessary overheads, and too infrequent polling might mean stale data.

This paper's contribution lies in the judicious combination of the two models. Today's web sources are "pull" based. Thus, we cannot use the server push technique to maintain consistency between the remote servers and our cache. Hence the cache must maintain its consistency by "pull"ing changes from the server. On the other hand, VW, i.e., the cache server, can be designed to pull data from the web servers and push them to the users.

Given this combination of `Client Pull` and `Server Push`, better performance can be obtained by updating a user's view only when the change is of interest to the user. That is, a user is allowed to specify consistency constraints, and we *push* new data to the user only when the change satisfies the constraints. For example, the user may specify that he is interested in a stock price change only when the price changes by at least a dollar. Note that even if the VW *pull*s data from remote servers when the change is less than a dollar, it *push*es the changed data to the user only when the change exceeds a dollar. When the user is remote to the VW, this feature further reduces the incurred Internet traffic overhead.

*To stay within the constraints imposed by today's web sources, we assume in the rest of the paper that these sources cannot be modified by us to "push" changes.* Also, *no change to the* `http` *protocol must be required.* So, Let us look at the possible ways in which judicious "pull"ing can be accomplished with *current* web infrastructure.

The web infrastructure gives us two types of "hooks" that can be useful.

1. `Time-To-Live` (TTL) values, attached to cached objects (HTML pages). Upon its expiration, the source of the object can be contacted to update the page.
2. A source can be contacted with an `if-modified-since` (a header field in a `http`) request [BLET95]. This causes the server to respond to the request only if the requested object has been modified since the specified time. If it has not been modified, the client continues to use the cached object, else it caches the new object.

Given this, the crucial issue is the setting of the TTL values for each cached object.

For minimizing the incurred network overheads, the value of TTL must be high. But a low TTL value may compromise temporal consistency. Thus any TTL value must be judged depending on two factors - how well the cache consistency is maintained, and how often the remote servers are polled. Ideally, we must dynamically update this TTL value using an algorithm that decides the value depending on the present and past rates of source changes, with the goal of keeping remote requests to a minimum while maintaining the needed temporal accuracy of the data. An algorithm that achieves this is presented in Section 3 and evaluated in Section 4.

The next Section gives an overview of the related work to place the issues addressed in this paper in perspective.

### 2.3. Prior Work

[ABGM90] is one of the earliest papers relating to the topic of maintaining coherency between a data source and cached copies of the data. This paper discusses techniques whereby data sources can propagate, i.e, push, updates to clients based on their coherency requirements. This paper also discusses techniques whereby cached objects can be associated with expiration times so that clients themselves can invalidate their cached copies.

More recently, various coherency schemes have been proposed and investigated for caches on the World Wide Web where the sources are typically pull-based and stateless. Thus, the source is unaware of users' coherency requirements and users pull the required data from the sources.

A *Weak consistency* mechanism, *Client polling*, is discussed in [CATE92], where clients periodically poll the server to check if the cached objects have been modified. In the Alex protocol presented here, the client adopts an adaptive Time-To-Live(TTL) expiration time which is expressed as a percentage of the object's age. Simulation studies reported in [JGMS96] indicate that a weak-consistency approach like the Alex protocol ([CATE92]) would be the best for web caching. The main metric used here is network traffic. While the Alex protocol uses only the time for which the source data remained unchanged, given our desire to keep temporal consistency within specified limits, we need to also worry about the magnitude of the change.

A *strong consistency* mechanism, *Server invalidation*, is discussed in [CLPC97], where the server sends invalidation messages to all clients when an object is modified. This paper compares the performance of three cache consistency approaches, and concludes that the invalidation approach performs the best.

Whereas our goal is to develop a method that does not entail any server modifications or changes to the `http` protocol, invalidation based protocols as well as other proposed protocols (described next) for maintaining cache consistency either require changes to the web sources or to the `http` protocol.

A survey of various techniques used by web caches for maintaining coherence, including the popular "expiration mechanism", is found in [ADTP96]. It also discusses several extensions to this mechanism, but,as discussed in [RCKR98], these do not meet our needs. Another approach is for the cache server to piggyback a list of cached objects [BKCW97] whenever it communicates with a server. The list of objects piggybacked are those for which the expiration time is unknown or the heuristically-determined TTL has expired. The paper discusses two approaches to implement this mechanism within HTTP/1.1 (i.e., without any changes to the protocol), but the servers must be modified to implement this mechanism. [BKCW98] discusses a similar approach, where the servers partition the set of objects into *volumes*, and maintain version information for each volume. When responding to clients, the server piggybacks a list of volume objects modified since the client-supplied version. Again, implementation of this mechanism requires changes to existing web servers.

## 3. Choosing a good TTL

Note that different stocks fluctuate at different rates and that a particular stock may have different dynamic behaviors with the passage of time. The implication of this for us is that the tracking of different stocks will require one TTL per stock price data and that the TTL associated with a particular stock is likely to change with time. Also, given the nature of stock prices, the server has no way to know until when a particular stock price will prevail. So we assume that the server does not suggest an expiration time for the stock price. It is worth mentioning that even if it did, given that different users may have different temporal coherency requirements, the fact that the data will change at a certain time may be less interesting to a user than the magnitude of the change.

We first present several candidate approaches to select TTL values and finally present an adaptive approach that proves to be very effective.

### 3.1. Static TTL – based on a priori assumptions

One obvious approach is to choose a low value of TTL and use it throughout, thus ensuring that cache data seldom gets stale. But the drawback of this approach is that a low TTL implies contacting the web servers too often, thus increasing network overheads. On the other hand, a high TTL may compromise cache consistency, although it reduces the network overhead. Thus a static TTL may not suffice. The only advantage of this approach is its simplicity, and this can be employed when source data changes are not rapid. However, for sources with time-varying data, a more dynamic TTL setting is necessary.

### 3.2. Semi-static TTL – based on observed maximum rate of change

One of the basic needs for maintaining temporal coherency is to be prepared to observe the quick changes that occur at the source. Suppose we do not know the rate at which changes occur at a source. A simple way to be prepared for quick changes is to adjust the TTL is to start with a large TTL and if the rate of change is more than can be observed by this TTL value, decrease the TTL value accordingly. This way, we can be prepared for the possibility of rapid changes at the sources.

Suppose $S(0)$, $S(1)$, …, $S(l)$ denote the data values at the source at different points of time in chronological order. That is, $S(l)$ is the most recent value. Define,

$$change_i = |S(i) - S(i-1)|$$

Let $T_0, T_1, \ldots, T_l$ denote the TTL values that resulted in the respective $W$ values.

Let the latest TTL value be $T_l$, and the latest data change, $change_l$. (Let the corresponding penultimate values be $T_{l-1}$ and $change_{l-1}$).

Let
$$TTL_{est_l} = (T_l / change_l) \times c$$

That is, $TTL_{est_l}$ is an estimate of the TTL value, based on the most recent observation, if we want to ensure that changes which are greater than or equal to $c$ are not missed.

Let $TTL_{mr}$ denote the most conservative TTL value used so far, i.e., the smallest TTL used so far. (This would have been set when the source changed rapidly.) This value is updated using

$$TTL_{mr} = Min(TTL_{mr}, TTL_{est_l})$$

This is the value of the new TTL. With this setting, the system is prepared even if the maximum rate of change observed so far recurs.

It is worth noting that the VW's computation of $TTL_{mr}$, based on the results of polling the source at specific points in time, is in fact an *estimate* of the maximum rate of change at the source. This is because, a stock price may change between two pollings at a rate higher than has been observed by the VW, but by the time the price is observed by the VW, the price comes down. This can happen if the TTL value is very large and hence the VW is unable to observe rapid changes in between two pollings. This suggests the need for setting TTL values which do not miss out on interesting changes and partly motivates the need to cap TTL values, as is done later. In general, a good understanding of the domain being observed, e.g., stock prices, will give indications about expected rate of change per unit time, given the constraints within which the domain operates. These can be used to set reasonable upper and lower bounds for TTL values.

Returning to the semi-static approach, it is pessimistic since it is based on the expected worst case rate of changes at the source. It will result in a large polling rate, especially if the worst-case does not occur often.

An alternative to the above semi-static approach is a dynamic approach wherein most recent changes, as opposed to the worst-case rate of change observed in the past, guide the selection of the TTL value.

### 3.3. Dynamic $TTL_{dr}$ – based on the most recent source changes

An alternative to static or semi-static TTLs is to assume a low TTL initially, and adjust the value depending on recent observations of source data changes. That is, if the source data changes very often, use a low TTL, and if the source data changes slowly, use a high TTL. It assumes that recent changes are likely to be reflective of the changes in the near future.

As before, let $TTL_{est_l}$ be a candidate for the next TTL value using only the most recent observations. Similarly, $TTL_{est_{l-1}}$ is a candidate for the next TTL value using only the penultimate observations[2]. $TTL_{dr}$, the new TTL value set by the *dynamic TTL* approach is given by

$$TTL_{dr} = (w \times TTL_{est_l}) + ((1 - w) \times TTL_{est_{l-1}})$$

where weight $w$ ($0.5 \leq w < 1$, initially 0.5) is a measure of the relative change between the recent and the old changes, and is adjusted so that we have the *Recency* effect, i.e., more recent changes affect the new TTL more than the older changes [KLAH72]. $w$ is computed as follows:

$$\delta = change_l / change_{l-1}$$

$$w = \begin{cases} \frac{\delta}{\delta+1} & \text{if } \delta > 1 \\ \frac{1}{\delta+1} & \text{otherwise} \end{cases}$$

Since $0.5 \leq w < 1$ (by definition), we always give at least half the weight to the estimate based on the most recent value. If the most recent change is much more (or much less) than the previous change, the weight $w$ gets closer to 1, thus giving a larger weight to the recent value. In this way, $TTL_{dr}$ is always in tune with the changes.

[2] It is important to note that although the method as shown here uses only two recent values of TTL, it can be easily extended to accommodate more values. Of course, this will mean that more history needs to be maintained by the VW.

### 3.4. Dynamic $TTL_{ds}$ – based on keeping TTL within static bounds

While the above adjustment of TTLs to derive dynamic TTL values appears to be a good idea, there is a subtle need that this approach does not satisfy: When source remains unchanged for a long period, the above algorithm will result in very large TTL values. To rectify this problem, we can use a static *interval*, and allow the TTL to vary within the interval. When the source changes rapidly, TTL values tend to get closer to $TTL_{min}$, the low end of the interval. During quieter times, TTL tends to move towards $TTL_{max}$, the high end of the interval. $TTL_{ds}$, the new TTL set by this algorithm is

$$TTL_{ds} = Max(TTL_{min},\ Min(TTL_{dr}, TTL_{max})).$$

The idea behind keeping the TTL value within a bounded interval is to allow the TTL to adapt to patterns of changes in source data but disallow it from assuming unreasonably low/high values.

### 3.5. An Adaptive approach

The adaptive algorithm uses the following factors to decide the new TTL.

1. Since we would like to keep TTL values bounded, $TTL_{min}$ and $TTL_{max}$ are used as static bounds.
2. Assuming that recent changes are likely to be reflective of the changes in the near future, $TTL_{dr}$ is a candidate for the new TTL value.
3. The system must be in a position to handle the worst case rate of change that has been previously observed (that may be higher than has been witnessed in the recent past). So $TTL_{mr}$ is also a candidate for the new TTL value.

With these in hand, the adaptive algorithm computes the new TTL to be,

$$TTL_{adaptive} = \\ Max(TTL_{min},\ Min(TTL_{max}, TTL_{dr}, TTL_{mr'}))$$

where, for reasons explained below, $TTL_{mr'}$, derived from $TTL_{mr}$, is used, instead of $TTL_{mr}$ itself.

As was noted before, if the recent change tends to zero, $TTL_{dr}$ will be very large, and before we contact the server next (when the large TTL expires), it is very likely that the source would have changed considerably. To avoid this, the estimated TTL is limited to be within $TTL_{min}$ and $TTL_{max}$. However, since $TTL_{max}$ is determined statically, it may not accurately reflect the current trend in source changes. This is the motivation behind $TTL_{mr'}$ defined as:

$$TTL_{mr'} = (f \times TTL_{mr}) + ((1 - f) \times TTL_{est_l})$$

where $0 \leq f \leq 1$ is the *fudge* factor. As defined earlier, $TTL_{mr}$ corresponds to the fastest source change so far, and $TTL_{est_l}$ corresponds to the recent change. Thus $TTL_{mr'}$ accommodates both of these, giving different weights to each of them depending on the $f$ factor. If $f$ is close to 0, we entirely rely on the recent trend; this will result in a loose upper bound if the recent source changes are slow. A high value of $f$ is preferable, because this gives more weight to a conservative TTL (corresponding to a period when source changes were the fastest). That is, once the source has changed rapidly, we believe that the source has the *potential* for future rapid changes. Use of $f$ allows us to control the pessimism that is unavoidable with the use of $TTL_{mr}$.

In summary, new TTL values are computed based on a combination of statically determined bounds, recent changes, and previously observed maximum rate of changes at the source.

Polling overhead incurred by the system, and the consistency guarantee provided critically depend on the *goodness* of the TTL values assigned to cached objects. Hence, an adaptive TTL (like ours) is expected to give considerable performance gains compared to the other candidate approaches. We next analyze the performance of the candidate algorithms and compare it with the performance of the adaptive algorithm.

## 4. Performance analysis

The algorithms introduced in the previous Section were evaluated using real world traces of stock prices to see how well they performed.

The presented results are based on stock price traces (i.e., history of stock prices) of a few companies obtained from `http://quote.yahoo.com`. We "cut out" the history for the time intervals listed in table 1 and experimented with the different mechanisms by determining the stock prices they would have observed had the source been live. This was done by noting the stock price in the trace at the time the "current" TTL expired. This is the price the source would have returned to the VW when contacted upon the expiry of the TTL, assuming that communication costs are negligible.

The graphs showing the performance metrics (defined in the next subsection) were obtained by averaging the results for these traces. We measured this average performance for different values of user constraint $c$, varying it from \$0.1 to \$0.7. The other graphs (TTL versus Time, and the graphs showing how the cache and the user display values follow the source) are based on the first trace from the list below. For these graphs, a value of \$0.6 was used as the user constraint.

Throughout this Section, the rate of change of the source is relative to the user constraint. For e.g., if the source changes at 10 cents a minute, it is *slow* relative to $c = \$0.9$, but is *fast* relative to $c = \$0.05$.

**Table 1. Traces used for the Experiment**

| Company | Date | Time |
|---|---|---|
| IBM | May 7, 1998 | 10:00-13:00 |
| IBM | Aug 6, 1998 | 10:00-13:00 |
| IBM | Aug 6, 1998 | 13:00-16:00 |
| Sun | Aug 6, 1998 | 10:00-13:00 |
| HP | Aug 6, 1998 | 10:00-13:00 |
| Sun | Aug 12, 1998 | 10:00-13:00 |
| IBM | Aug 12, 1998 | 10:00-13:00 |
| Microsoft | Aug 12, 1998 | 10:00-13:00 |

We now describe the metrics used and then present and analyze the results.

### 4.1. Metrics used

Performance is judged by how well a temporal coherency mechanism minimizes two metrics -

- *#pollings*: the number of times the source is polled – this metric is an indication of the networking overhead incurred by the algorithm.
- $VProb$: probability that a user's temporal coherency requirement is violated. $VProb$ indicates the probability that the user will not be notified about changes that exceed $c$ units (from what the user is currently aware of). It measures how well cache consistency is maintained relative to the requirement $c$.

While the first metric's semantics is clear, the second requires some explanation: In order to gauge how well cache consistency is maintained, we measure the duration for which the user is out of synch with the source data, when the price difference between source data and the value displayed to the user exceeds the user specified constraint. That is, $U(t)$ is different from $S(t)$, although $|U(t) - S(t)| \geq c$. Suppose this situation prevails continuously for a certain duration. Let $t_1, t_2, \ldots, t_n$ denote the durations when this happens. Let $T$ denote the total time for which data was presented to a user. Then the consistency violation probability is computed as,

$$VProb = \sum_{i=1}^{n} t_i \bigg/ T$$

and is expressed as a percentage. This then indicates the percentage of time when a user's desire to be within $c$ units of the source is not met.

The lower the $VProb$, the better the performance with respect to the cache consistency metric but higher the possible costs in terms of #pollings. There is clearly a tradeoff between the two metrics.

## 4.2. Performance comparison of the algorithms

Static TTL

One main attraction of a static TTL algorithm is its simplicity. If the rate at which the source changes is known, we can decide on a good TTL value, and use it throughout. But stock price variations are inherently unpredictable, and hence, choosing a good static TTL value is not trivial. Figure 2 shows how the cache and the user displayed values change with time when a static TTL of 16 minutes is employed. This means that the cache is refreshed every 16 minutes at which point the cache value is compared with the user value. If the difference is more than $c$ ($0.6 here), the user value is updated. If the source changes rapidly, this algorithm will not inform the user in time. This is what happens in the interval 75 to 95. Similarly, if the source changes very slowly, this algorithm will poll the source much more often than necessary.

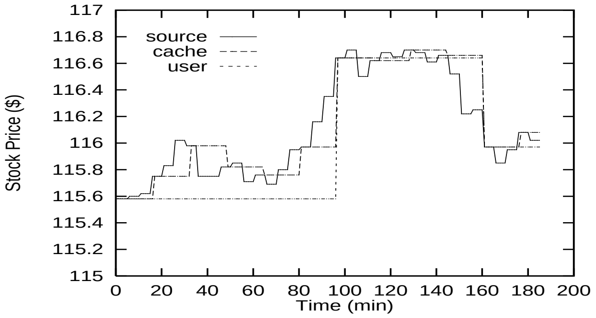


**Figure 2. Changes with time – Static TTL = 16 min**

In figure 2, the user requirement is violated around 90mins. But since the source is polled every 16mins, the violation is detected only at the next polling time which occurs at 96mins. A similar situation occurs between times 144 and 160.

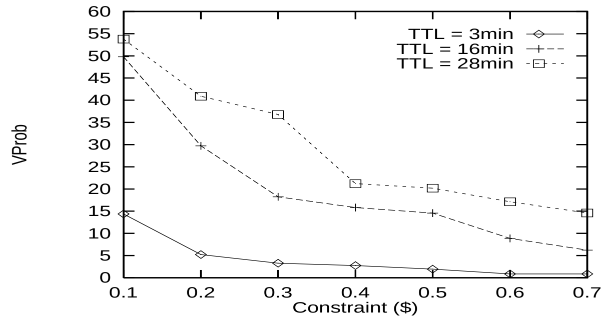


**Figure 3. Violation probability, Static TTL**

Figure 3 shows the performance of this algorithm for different values of static TTL. For low values (3 min), the curve for the probability of violation shows excellent performance, but the performance degrades with increasing TTL values (28 min). We see that the number of times the source is polled is very large for low static TTL and gets better as we use higher values. This algorithm is ideal when the rate of change of source is well known. For e.g., if we know that the source changes rapidly, we can use a low static TTL value. If the source changes slowly, we employ high values. But when the rate at which the source changes is variable and unknown (the typical case for stock prices), this algorithm can perform poorly.

With respect to #pollings, a Static TTL of 3 mins results in a total of 60 pollings whereas 16 and 28 mins result in 10 and 6 respectively. Since we strive to achieve better temporal consistency by choosing low static TTL values, the number of pollings (and hence the networking overheads) increases a lot. As we can see here, if we use a low static value for a relatively quiet source, we end up incurring unnecessary overheads.

Semi-static TTL

We do not analyze the performance of the semi-static algorithm because in some ways it is similar to static TTL, except that the TTL values drop lower when the source changes faster than it did ever before. So, in this method, TTL values can only get lower with time. That is, if a source changes fast initially, and much slower later, this method will waste significant bandwidth.

In particular, for low values of $c$, even small changes at the source make the source appear *fast*, and this method assumes low TTL values. It then behaves as in the previous method (with a low static TTL value). So, this method is not really useful for low $c$ values.

Dynamic TTL$_{dr}$, updates based on recent changes

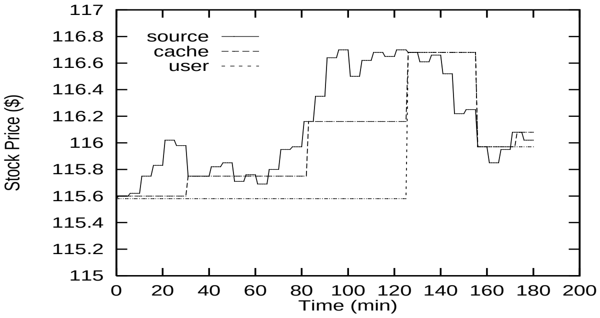


**Figure 4. Changes with time, Dynamic $TTL_{dr}$**

Figure 4 shows how the cache and the user displayed values change with time when this algorithm is used. The algorithm employs high TTL values when the source changes slowly. Because of the slow early changes, initial TTL values are high. (figure 7) . Because of this, the cache does not follow the source closely, and hence the probability of consistency violation is high (figure 5). Specifically, around

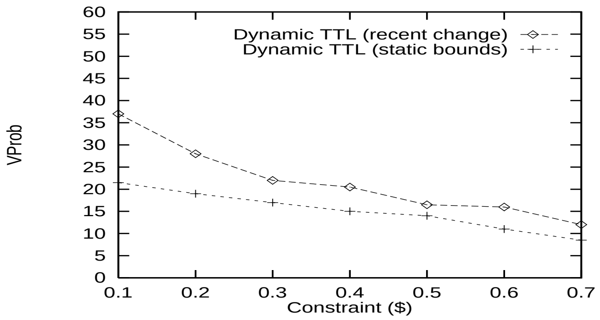


**Figure 5. Violation probability, Dynamic TTL**

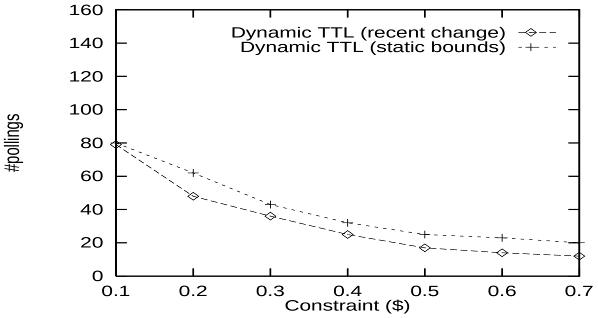


**Figure 6. #Pollings, Dynamic TTL**

80min, a TTL value of around 40mins is assumed (see figure 7). But in the next 40 mins, the source changes significantly, and the user is not informed even though the source has changed much more than \$0.6 (see figure 4) compared to the user displayed values. We encounter this problem because the source didn't change much initially, and the algorithm assumed high TTL values. The next algorithm aims to address this problem.

Figure 6 shows #pollings for this algorithm. For low $c$ values, most changes at the source appear fast. So, this method uses low TTL values, and hence, we have a greater number of pollings. For higher $c$ values, the performance of this approach in terms of the number of pollings is much better.

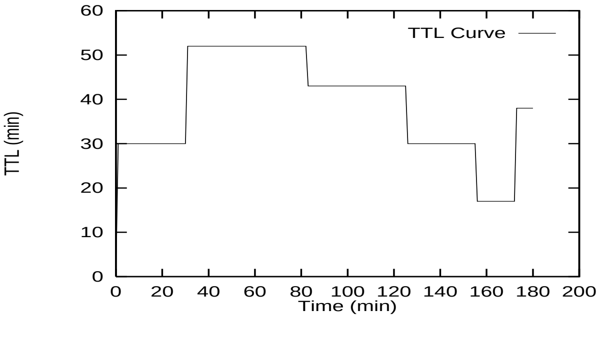


**Figure 7. TTL adaptation, Dynamic $TTL_{dr}$**

Dynamic TTL$_{ds}$, with static bounds ($TTL \in [1, 31]$)

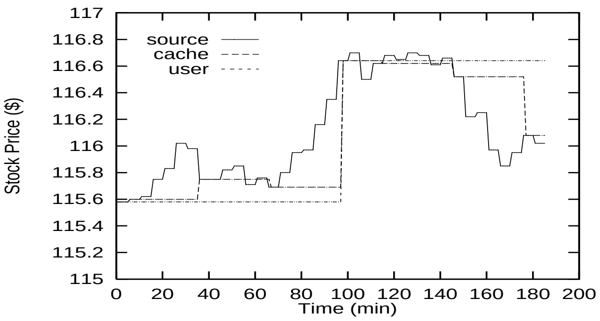


**Figure 8. Changes with time, Dynamic TTL$_{ds}$**

This is the same as the previous algorithm, except that we *statically* limit the TTL values. This is done to prevent the TTL values from assuming unreasonably high values. Depending on the rate of source changes, the TTL value will tend towards one end of the interval.

Figure 8 shows how the cache and the user displayed values change with time when this algorithm is used. When the system detects that the source changes rapidly, the cache is refreshed more often, and if the source is detected to change slowly, the cache is refreshed less frequently. As seen in figure 8, the user constraint is violated at around 90mins, and we detect this around 100mins. The upper bound on the TTL values helps us here; we would have assumed a higher TTL value around 50mins but for the upper bound (see figure 7). This helped us detect the violation earlier. Thus the upper bound solves, to some extent, the problem that surfaced in the previous algorithm. The next algorithm solves this problem to a greater degree.

Figures 5 and 6 show the performance of this algorithm. This algorithm performs worse than static TTL algorithm especially for higher 'c' values. This can be explained as follows. Initially, if the source changes slowly, the TTL value will assume the high end of the interval, and so the cache will not contact the source until this high value expires. If the high end of the interval was based on the slowest expected rate of change of the source, it is very likely that the source will change more rapidly when we assume a high TTL value. In such a case, the algorithm will fail to detect some changes. This explains the higher consistency violations recorded by this algorithm in figure 5. This observation is confirmed by the TTL adaptation curve shown in figure 9 where we see that the TTL value remains at the high end most of the time.

This algorithm is essentially the same as the previous method, except that we limit the TTL values. In the previous case, we found that the method assumed high TTL values and hence showed high probability of consistency violation. By using an upper bound, we have brought down the probability of consistency violation significantly (figure 5). The performance in terms of the number of pollings is not significantly more than the previous approach (figure 6).

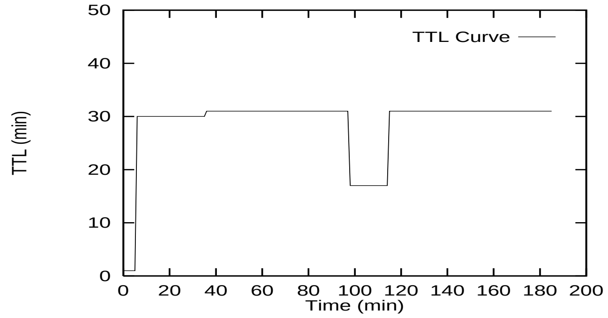


**Figure 9. TTL adaptation, Dynamic TTL$_{ds}$**

The algorithm's performance in terms of #pollings is comparable to the static TTL algorithm (with a static value of 16mins) for higher values of $c$, but the same cannot be said with respect to cache consistency maintenance.

Adaptive TTL

The problem with the previous algorithms was that we had to decide the TTL or the interval statically, depending on the expected patterns in source data changes. But typically, stock price changes are not predictable. In the adaptive algorithm, the key idea is to initially assume an arbitrary interval, and *update* not only the TTL value but also this interval depending on source data changes. (Presently, we alter the upper bound only.) We watch over the source and maintain the fastest rate of source change seen thus far. Using this, and the latest trend, we limit the upper bound. Since the latest trend could be in complete antithesis to rapid changes, it is better to give more weight to the bound based on the fastest rate. This can be achieved by using $f$ values close to 1.

We added another embellishment to the adaptive algorithm - to limit the rate at which TTL can increase. But instead of using additive increase (used in TCP/IP), we allow multiplicative increase, but limit the factor of increase. Figure 10 shows the performance of the algorithm for different $f$ values, when the new TTL is not allowed to be more than 2 times the previous estimate. The algorithm performs exceptionally well for higher values of $f$ (0.9), producing very low values of $VProb$.

Figure 11 shows how the cache and the user displayed values change with time when the adaptive algorithm (using a $f$ value of 0.9) is used. When the violation of the user constraint occurs around 90mins, this algorithm was working with a TTL value of only 9mins (see figure 12), and hence is able to detect this violation much earlier than the previous two algorithms. During the period from 100 to 140mins, the source changes very slowly, and the adaptive

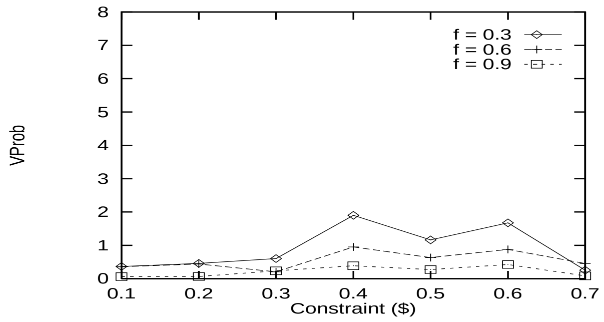


**Figure 10. Violation probability, Adaptive TTL**

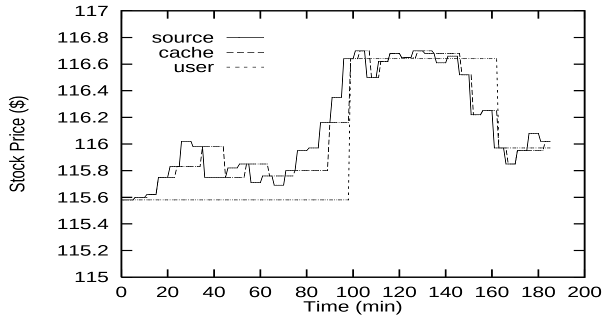


**Figure 11. Changes with time, Adaptive TTL**

algorithm assumes high TTL values (figure 12). But, the high TTL values are reached gradually in this period (instead of assuming high values suddenly), so as to not miss any potential change. (We limit the factor of increase to 2.) This way, the sudden decrease in the source value in the period from 140 to 160 mins is captured promptly by our algorithm. Note that in figure 12, there is a dip in this period. Thus the adaptive algorithm keeps a good watch on the source, without wasting too much bandwidth.

Figure 13 shows the #pollings of the adaptive algorithm for different $f$ values. The #pollings for this algorithm for a $f$ value of 0.3 is comparable to a static TTL of 3 min. But the consistency maintenance properties of this algorithm are much better than that of the static TTL (see figures 3 and 10). With higher $f$ values, we employ increasingly pessimistic methods (Section 3.2), and hence the #pollings increases. But the performance in terms of the probability percentage improves with increasing $f$ values (see figure 10). As we get more conservative, we use lower TTL values, and hence we get better consistency at the price of increased network overheads.

## 4.3. Summary of results

In Figures 14 and 15 we present an integrated view of the results presented so far. Not surprisingly, there is an almost inverse relationship between the number of pollings and the probability of violation.

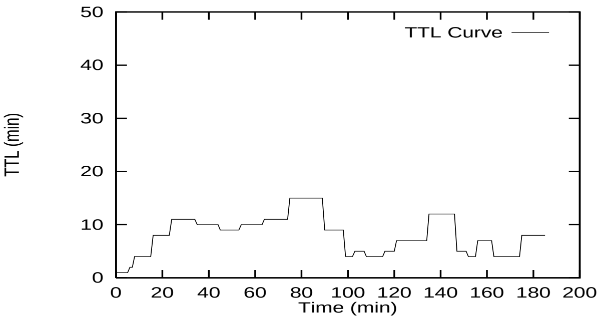


**Figure 12. TTL adaptation, Adaptive TTL**

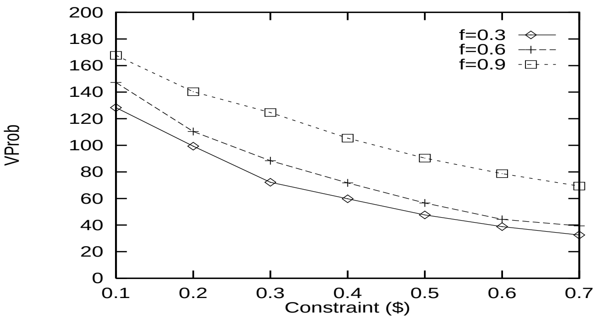


**Figure 13. #Pollings, Adaptive TTL**

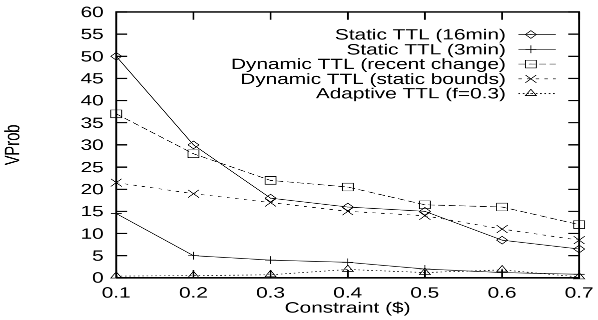


**Figure 14. Violation probability**

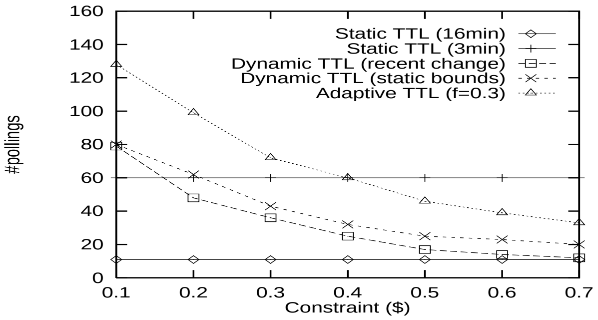


**Figure 15. #Pollings**

Adaptive TTL (with $f$=0.3, plotted in the graph, and for other $f$ values as well) has the best $VProb$. In fact, even if we were to continuously poll a source, given the communication costs involved in practice, there is bound to be a certain amount of temporal violation in practice. Given this, we believe that the violation probability values (of 0–2%) resulting from the use of Adaptive TTL represent perhaps the best one can achieve.

Low Static TTL (say, 3 mins) has the next best $VProb$ values. But, notice that Adaptive TTL not only performs better across all $c$ values tested, but it has a lower #pollings except for low $c$ values (i.e., below 0.3). When Static TTL values are increased (to, say, 16 mins), their performance degrades further to an extent that at low values of $c$ the Dynamic TTL algorithms are better.

Clearly, smaller static TTL values produce fewer temporal violations but result in a larger number of pollings for low to medium values of $c$. Even though Adaptive TTL also results in a large number of pollings, it produces a very low probability of violation.

In order to compare the relative cost of the superior performance of Adaptive TTL over Static TTL, we also computed the increase in cost (in terms of #pollings) per unit decrease in temporal violation (in terms of $VProb$). For example, at $c$ = 0.1, $VProb$ for Adaptive TTL is close to 0 whereas for Static TTL of 16 mins it is close to 50. This performance is produced at the cost of 120 (i.e., 130 - 10) additional pollings by Adaptive TTL. The cost per unit performance improvement is then 2.4 (i.e., 120/50). In fact, for different values of $c$ this factor lies between 2 and 4 when Static TTL of 16 mins is compared to Adaptive TTL. This seems to be a reasonable price to pay for the increased cache consistency resulting from Adaptive TTL.

We conclude that for data such as stock prices, whose rate of change is unpredictable, it is best to use an adaptive algorithm. But rather than basing the TTL estimate solely on the most recent trend, the past trend must also be taken into account. To avoid using large TTL values, thus potentially missing sudden source changes, it is good to use an upper bound to limit the TTL estimates. But since a static upper bound may be problematic for stock prices, it is better to use an upper bound that gets tighter when the source changes faster than before. Also, even if the source changes are slow initially, assuming a high TTL value may result in poor performance. Hence the adaptive method is preferable. By limiting the rate at which TTL can increase, Adaptive TTL gets even better performance.

## 5. Conclusion and Future Work

We discussed the issues involved in maintaining temporal coherency of a virtual warehouse. A combination of `Push` (by the proxy server) and `Pull` (by the clients) maintain data consistency, both between the remote sources and the proxy server's cache, and between the cache and the results displayed at the clients' end. Clients are allowed to specify temporal constraints, so that the displayed results are updated only when the changes are of interest to the user.

We presented several approaches for setting TTL values and analyzed their performance. The adaptive algorithm's performance was shown to be much better than other algorithms that are less adaptive. We used two main factors in evaluating performance - how well cache consistency is guaranteed, and how many times the system contacts the data source.

There are several embellishments to the adaptive algorithm that we plan to explore. Presently, the adaptive algorithm keeps track of the TTL corresponding to the maximum rate of change at the source so far. This means that once the source changes rapidly, future TTL values will be quite conservative – and it gets tighter and tighter, when the source changes more rapidly than before. Suppose we believe that more recent trends are better indicators than trends indicated by $all$ of history. Then the pessimism that is currently displayed by $TTL_{mr}$ can be overcome by maintaining $TTL_{mr}$ based on a fixed time into the past, instead of basing it on the entire past.

Also, currently, we update only the high end of the TTL interval. To obtain better performance in terms of the number of pollings, it is desirable to update the lower end of the interval as well (using the rate at which the source changes). We plan to explore these issues as part of our future work.

Finally, we are currently implementing a Virtual Warehouse architecture based on the ideas proposed so far. In it, data is brought from remote sources and maintained in the (cache of) the VW. Data from this cache is used to serve the needs of users who are connected to the VW via a network. The system consists of two main modules - the cache server, and the data manager. The cache server is responsible for getting data from the remote web sources, maintaining the cache, and informing the data manager when data in cache gets updated. The data manager is responsible for getting input from the user and updating the displayed results whenever user specified constraints have been satisfied. While there is one data manager per user, the cache server is common to all users. (Details of the implementation can be found in [RCKR98].)

With this implementation in hand, we plan to conduct experiments and make measurements of message overhead, processing overhead and actual performance of our mechanisms. We also plan to develop techniques to efficiently schedule the VW's requests that pull different pieces of information from web servers once the associated TTLs expire. This is especially of interest for those servers that are connected to the VW via a single link.


## 6. Acknowledgment

We thank Jennifer Rexford for her constructive comments on previous versions of the paper. We also thank the anonymous reviewers for their suggestions.


## References


[ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina, Data Caching Issues in an Information Retrieval System, *ACM Trans. Database Systems,* September 1990.

[BLET95] T. Berners-Lee, Hypertext Transfer Protocol HTTP/1.0, *HTTP Working Group Internet Draft*, October 14, 1995.

[CATE92] A. Cate, Alex - A Global Filesystem, *Proceedings of the 1992 USENIX File System Workshop, Ann Arbor,MI*, May 1992.

[ADTP96] A. Dingle and T. Partl, Web Cache Coherence, *Proc Fifth Intnl WWW Conference*, May 1996.

[JGMS96] J. Gwertzman and M. Seltzer, World-wide Web cache consistency, *Proceedings of the 1996 USENIX Technical conference, San Diego, CA*, January 1996.

[KRGL97] M. Kamath, K. Ramamritham, N. Gehani, and D. Lieuwen, WorldFlow: A System for Building Global Transactional Workflows, *Proc. of 7th Intl. Workshop on High Performance Transaction Systems (HPTS)*, 1997.

[KLAH72] A. Klopf, Brain Function and adaptive systems- a heterostatic theory,*Tech Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA*, 1972.

[BKCW97] B. Krishnamurthy and C. Wills, Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web, *Proc. USENIX Symp. on Internet Technologies and Systems*, December 1997.

[BKCW98] B. Krishnamurthy and C. Wills, Piggyback Server Invalidation for Proxy Cache Coherency, *Proc. World Wide Web Conference*, April 1998.

[CLPC97] C. Liu and P. Cao, Maintaining Strong Cache Consistency in the World Wide Web, *Proceedings of ICDCS*, May 1997.

[KRAM93] K. Ramamritham, Real-Time Databases, *Journal of Distributed and Parallel Databases*, Volume 1, Number 2, pp.199- 226., 1993.

[RCKR98] R. Srinivasan, C. Liang, K. Ramamritham, *Maintaining Temporal Coherency in Virtual Warehouses*, University of Massachusetts, Amherst, Technical Report, September 1998.

[HZFJ98] H. Zou and F. Jahanian, Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees, *Proceedings of ICDCS*, May 1998.