Maintaining Temporal Coherency of Virtual Data Warehouses[1]

Raghav Srinivasan, Chao Liang and Krithi Ramamritham
Department of Computer Science
University of Massachusetts
Amherst MA 01003
{raghav,cliang,krithi}@cs.umass.edu

Abstract

In Electronic Commerce applications such as stock trading, there is a need to make decisions based on information collected from various sources. Many of these sources are accessible from the World Wide Web. Because the information such as stock prices keep changing, the web sources must be queried continually to maintain *temporal coherency* of the data in the *virtual warehouse* 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 discusses the maintenance of temporal coherency of data in a virtual warehouse, using stock trading data as the motivating example.
Keywords: Electronic Commerce, Temporal Correctness, Data warehouses, World Wide Web.

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 needed 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.

A challenging issue here is to maintain VWs up to date as updates occur at the data sources. What is needed 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 holder 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 the 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.

The rest of the paper is organized as follows - Section 2 explains our system architecture in detail.

[1]Supported by NSF grant IRI-9619588.

Section 3 discusses different ways of determining *when* data sources must be contacted to obtain up-to-date data values so as to meet user-level temporal coherency requirements, and provides details of our proposed algorithm. Section 4 analyzes the performance of our algorithm using real-world stock market data streams. In section 5, summarizing remarks and the future directions of our work are presented.

2 Maintaining Temporal Coherency

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 and serve the needs of the different clients from this VW.

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. This is especially true of the more dynamic (i.e., volatile) data such as stock prices. This *deviation* of temporal coherency needs to be kept within user-specified bounds. We must employ data refreshing schemes by which stock prices can be shown dynamically to the brokers as prices change. We propose to achieve timely updates to the 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.

In the rest of this Section, after giving an overview of an architecture for meeting the above needs, we discuss the issues in designing each of the components of the architecture, and then provide details of the architecture.

2.1 Overview of an Architecture for the Virtual Warehouse

We are currently developing a Virtual Warehouse architecture. 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. As shown in figure 1, our 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. 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 my stock price every 5 mins.* (Later we'll show how constraints in the value domain can be translated into constraints with respect to the time domain, i.e., into requirements on *when* data at a client needs to be updated.) While

there is one data manager per user, the cache server is common to all users.

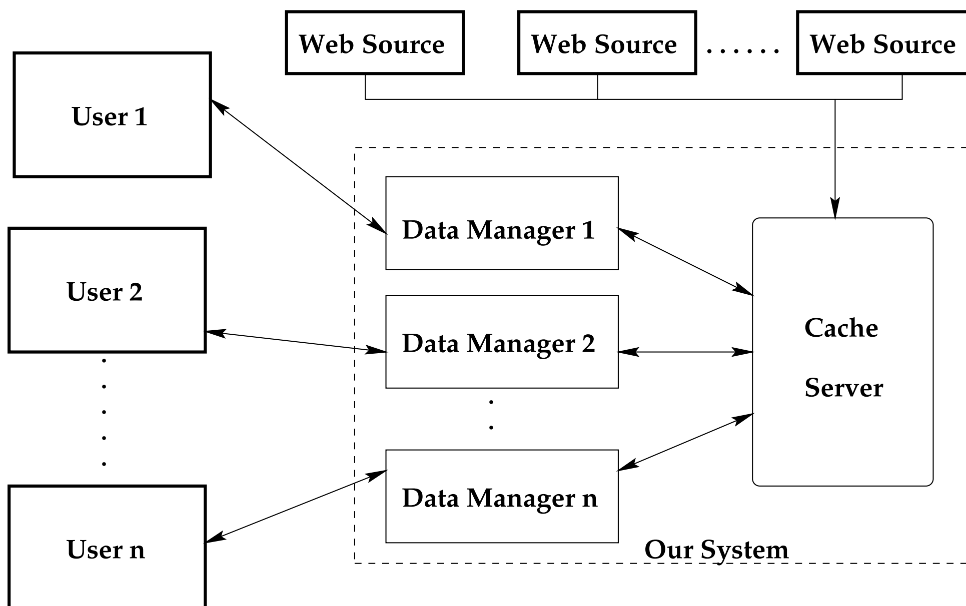

Figure 1: System architecture

2.2 Issues in keeping the Virtual Warehouse Temporally Coherent

Temporal coherency is concerned with the relationship between an object in the real world and its image in a database [KRAM93, HZFJ98]. For the purposes of this paper, the state of the real-world objects is maintained in the web servers, and the images are maintained by the cache server, and also in the user's view. To a user, the cache server acts as the server and to the cache server the web sources act as the servers.

In a typical client-server environment, maintaining consistency of the client and the server can be achieved in one of two ways - `Client Pull`, or `Server Push`. In the former, the client *pulls* data from the source (whenever it suspects that data might have changed at the source). In the latter, the source of the data, i.e., the server, *push*es data to the client (whenever data changes at the source).

The consistency guarantee of the `Client Pull` mechanism critically 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. A balance ought to be struck.

In our system, the cache server maintains the cache and is the server for the clients. Here we can use the `Server Push` mechanism to effect changes in the results displayed for the clients.

Note that we cannot, however, use the same technique to maintain consistency between our cache and the remote Web sources, since Web sources today work based on the "pull" approach and cannot be modified by us to "push" changes. Hence the cache must maintain its consistency by "pull"ing changes through judicious use of TTL values.

Although this combination of `Client Pull` and `Server Push` mechanisms significantly reduces the incurred networking overhead, better performance can be obtained by updating the results only when the change is of interest to the user. That is, the 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 our cache may still *pull* data from remote servers even when the change is less than a dollar, but our system *push*es the changed data to the user only when the change exceeds a dollar. Thus, when the user is remote to our system, this feature further reduces the incurred Internet traffic overhead.

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 requests 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. The idea is to 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. In the rest of this Section, we present the details of the VW components.

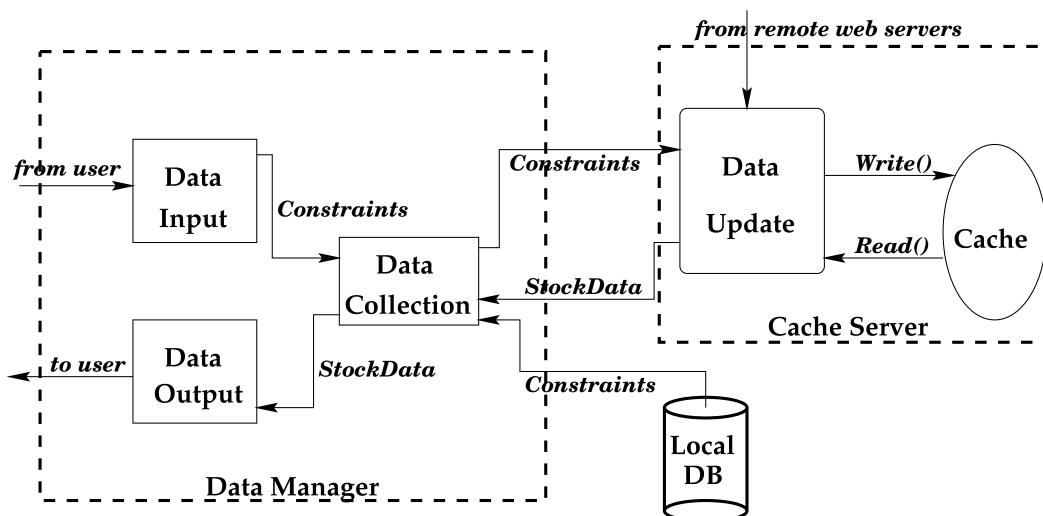

Figure 2: Detailed architecture of our system

As shown in figure 2, the data manager has three main modules: *Data Collection (DC)* module, *Data Input (DI)* module, and the *Data Output (DO)* module. *DI* and *DO* essentially perform data format conversions, *i.e.,* changing user input into a form comprehensible to *DC* and *vice versa.* They also perform the task of managing the user input form. The list of companies (whose stocks the brokerage firm is interested in), consistency constraints, company information, etc, are available in a local database (typically in a disk). The *DC* module is responsible for relaying the user requests to the cache server, and collecting results from the cache server. It also maintains one copy of the data currently displayed to the user. Whenever it gets updated data from the cache server, it uses this data to check if the user specified constraints have been satisfied and pushes this data to the client accordingly. Note that the *DC* cannot update its previous copy unless it sends the new data to the user. Otherwise, two or more changes which cumulatively satisfy the user constraint (but no two *successive* changes satisfy the constraint) will *not* be reported to the user, thus compromising temporal correctness requirements.

The cache server acts as a proxy too. All user requests are relayed to our cache server by data managers. On receipt of a request from the data manager, the *DU* module checks the cache. If the requested company information is available in the cache, data is read from the cache and passed onto the data manager. Otherwise, it fetches data from the remote Web servers and passes it to the data manager, while caching the data. Specifically, the *Data Update (DU)* module is responsible for maintaining cache consistency. It is also responsible for informing the data managers interested in a company's stock when the company's data gets updated in the cache. Thus our system uses the cache `invalidation` message to notify the data managers. The data managers then contact the cache server to get the updated data. *DU* is thus an *active* module, but the cache remains *passive.* The cache can supports `read()` and `write()`, and is completely hidden from the data manager.

If two different users of our system specify the *same* company but *different* constraints, our cache stores only one copy of the company's data (*DU* uses the *stricter* constraint). But for each user, we will have an exclusive *DC* module, which checks for the user's constraints whenever the cache is updated.

The *DU* module uses the `Client Pull` mechanism to maintain cache consistency. It assigns a `time-to-live` value (TTL) for each cached object. Once the TTL expires, *DU* fetches data from the remote Web servers again, using the `if-modified-since` header field of `http`.

For minimizing the incurred network overheads, the value of TTL must be low. But a low TTL value may compromise cache 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. The next section discusses this crucial issue of arriving at a good TTL value.

3 Choosing a good TTL

We first present several candidate approaches to select TTL values and then present our approach.

3.1 Candidate approaches

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. A high TTL may compromise cache consistency, although it reduces the network overhead. Thus a static TTL will never 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.

To this end, an alternative approach is to assume a low TTL initially, and *adapt* it depending on patterns in 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. While this appears to be good, there is a subtle need that this approach does not satisfy: When source remains unchanged for a long period, this algorithm will start using very large TTL values. If the source begins to change suddenly, the system will not contact the server until the large TTL expires, and hence the cache will no longer be consistent.

A good way to accommodate such a situation is to bound the TTL values. Instead of using a static TTL, we can use a static *interval*, and allow the TTL to vary within the interval. TTL values tend to get closer to the low end of the interval when the source changes rapidly. During quieter times, TTL tends to move towards the high end of the interval.

Both these approaches, static TTL and a static interval for TTL, raise an important question - how does one decide the TTL value or the TTL range? Unless we use a good value or a range, temporal coherency can suffer or overheads may be unnecessarily high.

The idea behind using a bounded interval is to allow the TTL to adapt to patterns of changes in source data. One can extend this logic to use the change patterns to decide the interval range itself. So, TTL values vary within a *dynamic* interval, with the TTL value and the interval being determined by observing the dynamics of the changes in source data.

We next discuss an algorithm that uses the last approach. We then discuss the performance of this algorithm, comparing its performance with the first two approaches.

3.2 Exploiting dynamics of the source data - our algorithm

Let us focus on the three main components of our architecture (See 3).

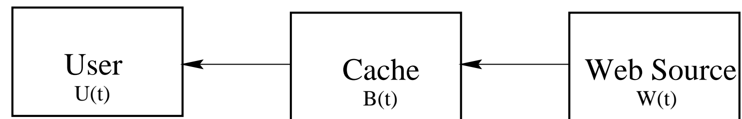

Figure 3: Components of our system where the object and their images lie

Here, $W(t)$ denotes the data source value, $B(t)$ 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 our system, 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.

A good algorithm must guarantee,

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

Suppose the source is *sampled* at specific points in time. Let $W(0)$, $W(1)$, ..., $W(l)$ denote the data source values of the samples in chronological order. That is, $W(l)$ is the most recent value. Define,

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

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

The DU module maintains the latest TTL value, T_l, and the latest data change, $change_l$. The corresponding penultimate values T_{l-1} and $change_{l-1}$ are also maintained.

Thus our algorithm uses the following factors to decide the new TTL:

1. A statically specified minimum (TTL_{min}) and maximum (TTL_{max}) TTL values are associated with each data item. These are based on an aggregate view of the source data and the nature of the changes. The idea is to ensure that TTL values used are within reasonable limits.

2. Recent changes are likely to be reflective of the changes in the near future. To this end, the new candidate TTL (TTL_{cand}) is computed based on the two most recent TTL values and the changes that were detected with these values [2].

3. The system must be in a position to handle changes that occur at a rate that has been witnessed in the past. To this end, a new upper bound for the TTL ($TTL_{max'}$) is computed based on the present rate of source change and on the maximum rate of source change so far.

With these in hand, our algorithm decides the new TTL to be,

$$TTL_{new} = Max(TTL_{min}, Min(TTL_{cand}, TTL_{max}, TTL_{max'}))$$

We now provide details of how TTL_{cand} and $TTL_{max'}$ are determined.

Let

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

$$TTL_{est_{l-1}} = (T_{l-1} / change_{l-1}) \times c$$

TTL_{est_l} is an estimate of the next TTL value using only the most recent observations. Similarly, $TTL_{est_{l-1}}$ is an estimate of the next TTL value using only the penultimate observations.

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

[2] It is important to note that although this 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 the DU module must maintain more history.

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 recent values. 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_{cand} is always in tune with the changes.

As we noted before, if the recent change tends to zero, our estimate of TTL 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, we limit the estimated TTL 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 $TTL_{max'}$ described next.

Let $TTL_{max-rate}$ denote the most conservative TTL value used so far, *i.e.,* the least estimate so far. (A low estimate of TTL means that we expected the source to change rapidly.) This value is updated using

$$TTL_{max-rate} = Min(TTL_{max-rate}, TTL_{est_l})$$

.

$TTL_{max'}$ can now be computed as follows:

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

where $0 \leq f \leq 1$ is the *fudge* factor. $TTL_{max-rate}$ corresponds to the fastest source change so far, and TTL_{est_l} corresponds to the recent change. Thus $TTL_{max'}$ accommodates both of these, giving different weights to each of them depending on the f factor. The value of f determines the upper bound used for changing the high end of the interval. If we use a value 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.

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

Networking overhead incurred by our system, and the consistency guarantee provided by our system critically depend on the *goodness* of the TTL values assigned to cached objects. Hence, an adaptive TTL (like ours) gives considerable performance gains compared to a static TTL. We next analyze the performance of our algorithm, comparing it with static TTL, and a TTL that varies within a static interval.

4 Performance analysis

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

The presented results are based on IBM stock prices observed on May 1, 1998, from 10AM to 1PM (as obtained from the data source `http://quote.yahoo.com`). The overall conclusions resulting from this data source stream were true of other stock price streams as well.

A value of \$0.6 was used for the user constraint c. 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 $c = \$0.05$.

4.1 Metrics used

The performance is judged by two factors -

- the number of times the source is polled – this metric is an indication of the networking overhead incurred by the algorithm. We would like to minimize it as much as possible.
- how well cache consistency is maintained relative to requirement c. This is measured as a probability that a user's requirement – to be notified about changes that exceed c units (from what the user is currently aware of) of a source – is violated.

In order to measure 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 $W(t)$, although $|U(t) - W(t)| \geq c$. 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 probability is computed as,

$$Prob = \sum_{i=1}^{n} t_i \Big/ T$$

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 $Prob$, the better the performance with respect to the cache consistency metric but higher the possible costs. There is clearly a tradeoff between the two metrics.

4.2 Performance comparison of the three algorithms

Case 1 : 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.

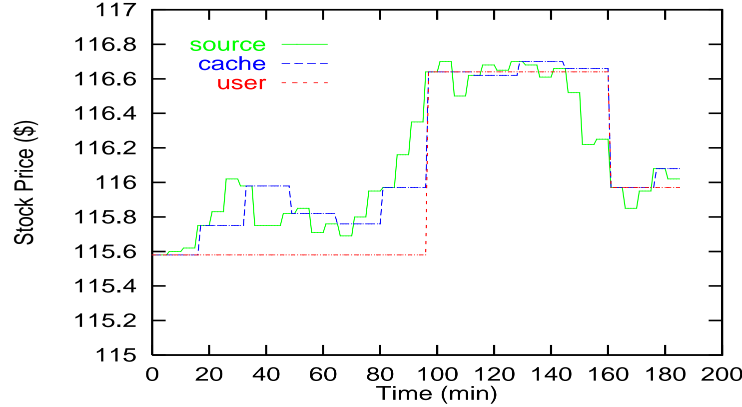

Figure 4: Data value changes with time

Figure 4 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, and the cache value is then 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. Similarly, if the source changes very slowly, this algorithm will poll the source much more often than necessary.

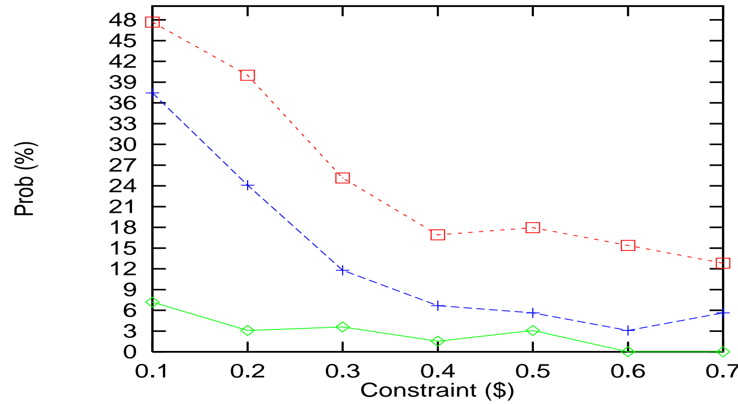

Figure 5: Prob of violation

Figure 5 shows the performance of this algorithm for different values of TTL. For low values (3 min), the curve for the Prob of violation shows excellent performance, but the performance degrades with increasing TTL values (28 min). Networking overheads (measured in terms of the number of times the source was polled) are very large for a 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.

Case 2 : Static Interval$(1min \leq TTL \leq 31min)$

To avoid using a fixed TTL for the entire period, thus remaining insensitive to drastic changes in

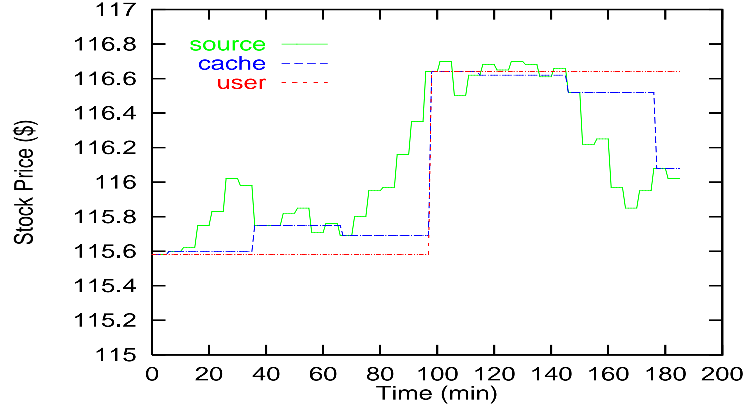

Figure 6: Data value changes with time

source change rates, one can allow the TTL to range within a predefined interval. This is accomplished using the procedure that was used to compute TTL_{cand} in the previous section – but bounded by the interval. Depending on the rate of source changes, the TTL value will tend towards one end of the interval. Again, similar to the issue in the previous algorithm, we can decide on a good interval if the patterns in source changes are known. For stock price variations, choosing a good interval may not be trivial.

Figure 6 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.

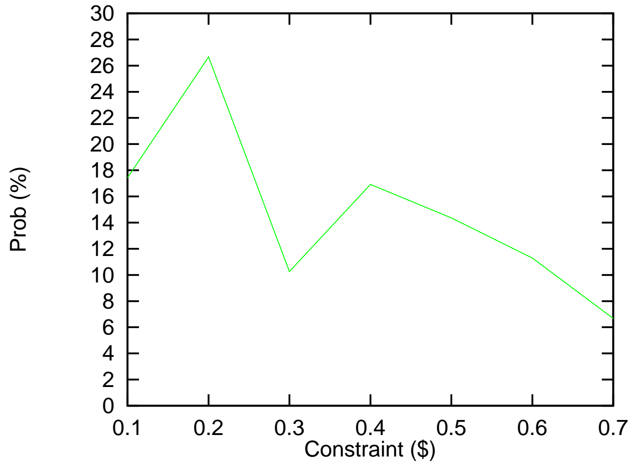

Figure 7: Prob of violation

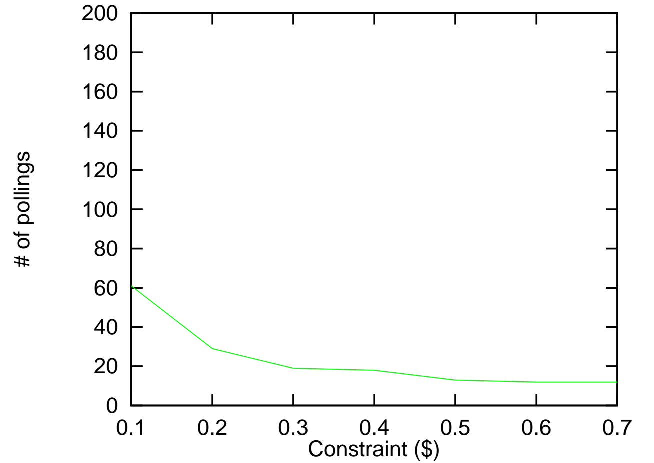

Figure 8: Polling frequency curve

Figures 7 and 8 show the performance of this algorithm. Contrary to one's expectations, this algorithm performs worse than the previous one (static TTL). This can be explained as follows. Initially, if the source changes slowly, the TTL value will assume the high end of the interval, and hence, 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 report to the user, when it should have. This explains the high average values recorded by this algorithm in figure 7. Our observation is confirmed by the TTL adaptation curve shown in figure 9. Note that the TTL value remains at the high end most of the time.

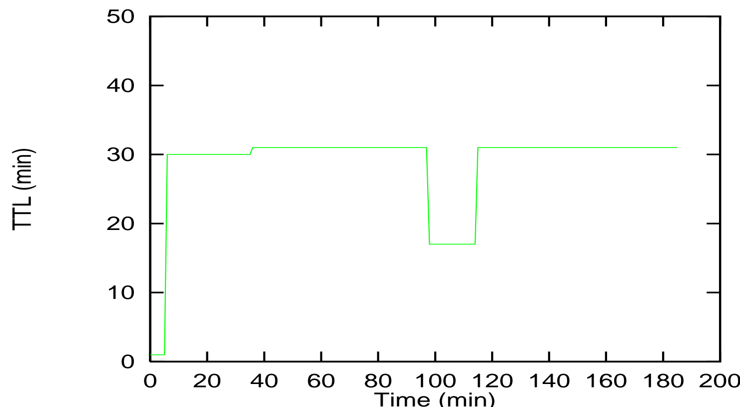

Figure 9: TTL adaptation

The algorithm's performance in terms of number of source contacts (the first metric) is better than the static TTL algorithm, but the cache consistency maintenance properties of this algorithm are very poor.

Case 3 : Dynamic Interval - our algorithm

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 our 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. Figure 10 shows the performance of our algorithm, for different values of f. Higher f values show better performance. But still, the performance is not as good as expected.

The poor performance can be explained from the observation that it is important to monitor the source for at least a while before deciding to use a high TTL. A good heuristic is 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 11 shows the performance of the modified algorithm for different f values, using a factor of 2 (that is, no TTL estimate can be more than 2 times the previous estimate). The modified algorithm performs exceptionally well for higher values of f (0.9).

For low constraints, the polling frequency curve (figure 14) shows a high value. This can be improved by dynamically adjusting the lower bound of the interval as well.

Figure 12 shows how the cache and the user displayed values change with time when our algorithm

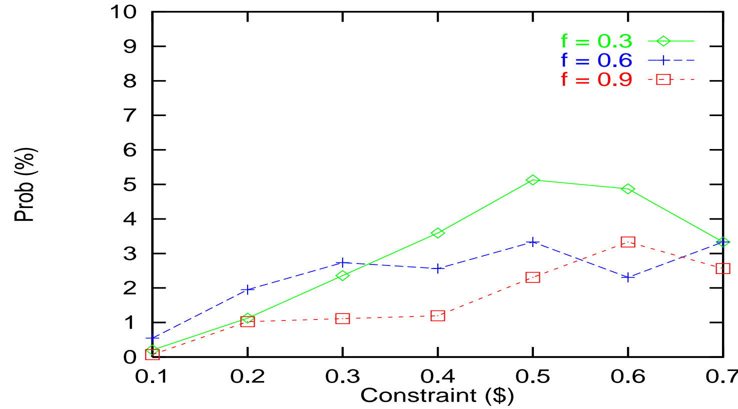

Figure 10: Prob of violation

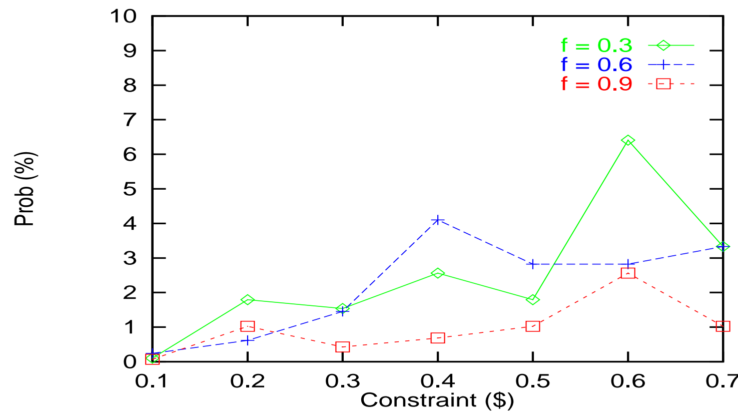

Figure 11: Prob of violation

(using a f value of 0.9) is used. Note how the cache closely follows the source, although the number of source contacts for a c value of 0.6 is only about 25 during the interval observed, which is as good as a static TTL of 8 min.

Figure 13 shows how the TTL adapts to changes at the source. Note how the TTL dips when source starts changing rapidly. Also note that the increases are gradual, not sudden.

Figure 14 shows the performance of our modified algorithm using the first metric, *i.e.*, the number of source contacts. The high values recorded for low values of user constraints can be minimized by dynamically updating the low end of the interval as well (using source change rates).

4.3 Summary of results

From this analysis, 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. Since a static upper bound was shown to be no good for stock prices (see figure 7), it is better to use

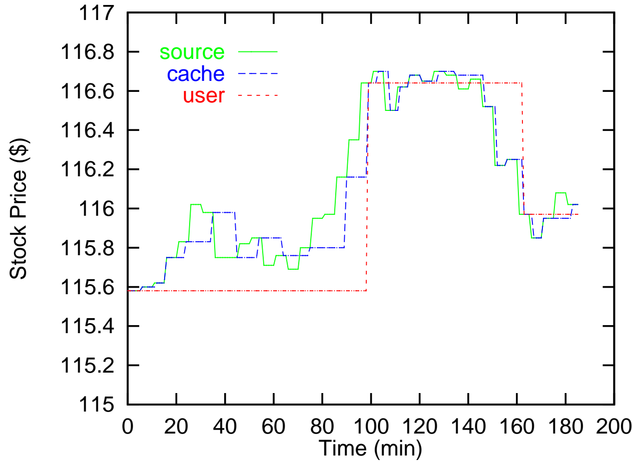

Figure 12: Data values with time

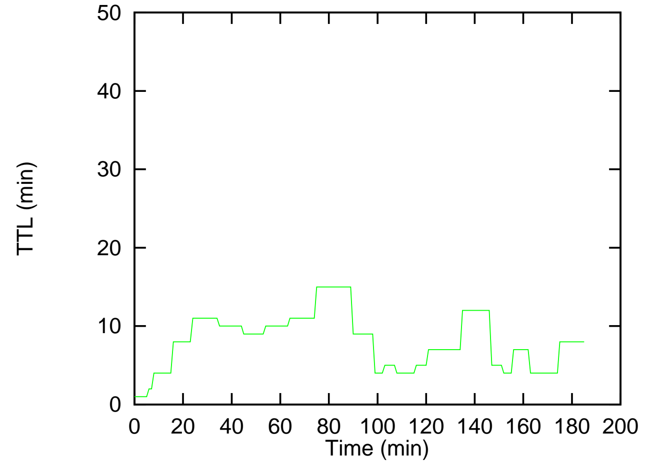

Figure 13: TTL adaptation

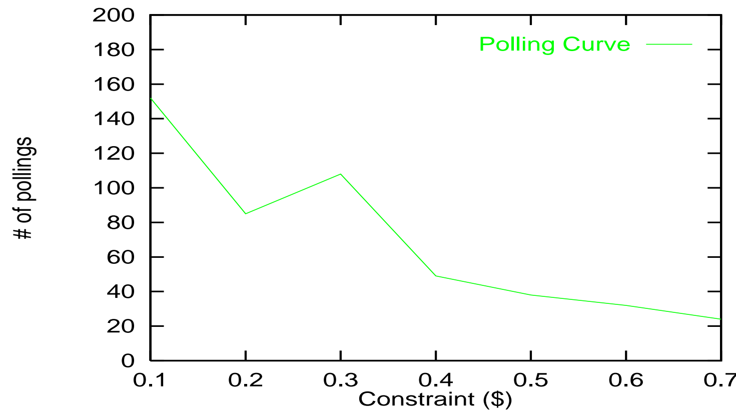

Figure 14: Polling Frequency Curve

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. By limiting the rate at which TTL can increase, we get even better performance (see figure 11).

5 Conclusion and Future Work

We discussed the issues involved in maintaining temporal coherency of a virtual warehouse and presented an architecture we are developing for this purpose. The architecture is designed to minimize network traffic and provide data consistency effectively. 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 an algorithm for dynamic TTL updates and analyzed its performance. The 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 traverses the Net.

Cache consistency algorithms for the World Wide Web are discussed in detail in [JGMS96, CLPC97]. But none of them talk about an adaptive TTL like ours, which decide TTL values based on patterns in source data. [CATE92] discusses a method which decides the TTL based on the cached object's age, but that will not suffice for our purpose. This is because, if the source remains unchanged for extended periods, Alex protocol will start using high values for TTL, unlike in our case where the TTL is bounded.

Presently, the 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, the upper limit comes down. In our algorithm, the upper limit never gets relaxed. In gets tighter and tighter, when the source changes more rapidly than before. In case when a source changes very slowly after a period of rapid updates, it is better to increase the upper limit of the interval, so that source contacts are minimized. This can be accomplished by maintaining $TTL_{max-rate}$ based on a fixed time into the past, instead of basing it on the entire past.

Currently, we update only the high end of the interval. To obtain better performance in terms of polling frequency, 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.

References

[CLPC97] Chengjie Liu and Pei Cao, Maintaining Strong Cache *Consistency in the World-Wide Web, In Proceedings of ICDCS'97, pp. 12-21, May 1997.*

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

[CATE92] Cate, V, Alex - A Global Filesystem, Proceedings of the 1992 USENIX File System Workshop, Ann Arbor,MI,May 1992,1-12.

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

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

[HZFJ98] Hengming Zou and Farnam Jahanian, Real-Time Primary-Backup (RTPB) Replication with Temporal Consistency Guarantees, ICDCS, 1998.

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

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