A Slacker Coherence Protocol for Pull-based Monitoring of On-line Data Sources*

Radhakrishnan Sundaresan[‡], Tahsin Kurc[†], Mario Lauria[‡], Srinivasan Parthasarathy[‡], and Joel Saltz[†]

[‡] Dept. of Computer and Information Science
The Ohio State University
Columbus, OH, 43210

[†] Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210

{sundarer,lauria,srini}@cis.ohio-state.edu
{kurc-1,saltz-1}@medctr.osu.edu

Abstract

An increasing number of online applications operate on data from disparate, and often wide-spread, data sources. This paper studies the design of a system for the automated monitoring of on-line data sources. In this system a number of ad-hoc data warehouses, which maintain client-specified views, are interposed between clients and data sources. We present a model of coherence, referred to here as slacker coherence, to address the freshness problem in the context of pull-based protocols. We experimentally examine various techniques for estimating update rates and polling adaptively. We also look at the impact on the coherence model performance of the request scheduling algorithm at the source.

1 Introduction

One of the consequences of the growth of the Internet, and in particular the world wide web, is the explosion of distributed online applications that operate on disparate, and often wide-spread, data sources. The immediate availability of such extensive amount of on-line information has the potential of enabling discovery through data analysis. For example, important knowledge on several types of cancer can be derived by correlating data from gene expression data, from tissue banks, and from drug sensitivity data of cancerous cell lines. Catastrophy prevention and remediation could be enhanced by the combined analysis of multiple sources of sensitive data such as weather satellite data, power grid utilization patterns, phone switches loads.

In a distributed environment, a client should be able to formulate queries that can span large data sets at multiple data sources. Moreover, the client should be informed of updates to the data of interest. In a subscriber/publisher system, the client can register to various data sources and specify the data of interest. The data sources are responsible for sending the updates to the data of interest to the client. In this way, the updates are pushed to the clients through mechanisms such as broadcast or multicast. With the emergence of Web services and Grid services technologies, we are seeing a shift towards a services oriented view of the Internet and the Grid. These technologies provide standard mechanisms for individuals or groups to make services and data available to a larger community and for enabling interoperability among various services. However, they do not dictate any requirements as to what the characteristics of the services should be. As a result, a client has to be able to interact with services that have different capabilities.

In such a setting, substantial practical obstacles limit our capability to take full advantage of large volumes of data. First, data sources can employ some form of RPC mechanisms that work well for "function shipping"—moving the process to the data—but they do not work well for data movement initiated by the data source. Using message passing requires the client to implement ad-hoc communication and coherence protocols in order to manage data copies. Second, existing models do not provide efficient mechanisms to inform the client of data changes. This puts the onus on the client to poll a data source (resulting in an ad-hoc non-deterministic communication pattern). Finally, the sheer amount of data and data sources makes a manual approach to data retrieval impractical, while at the same time existing protocols are oriented to human interaction (where a pull-only model is quite satisfactory).

In this paper we study the design of a system for the automated monitoring of on-line data sources. In our system a number of ad-hoc data warehouses are interposed be-

*This work was supported in part by an Ameritech Faculty Fellowship grant, the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497).

tween clients and data sources. The data warehouses have the tasks of i) maintaining client-specified views, defined over the data made available by a set of data sources, and ii) keeping the views updated as the source data changes over time. We are interested in designing a system that can work with standard Web protocols, so as to make it immediately usable with existing web-based data sources. Maintaining the freshness of views when using a pull model prevalent in current standards requires some form of polling. There are obvious trade-offs between freshness and resource utilization in deciding the polling frequency. The polling strategy must take into account the number of data sources involved in maintaining a view, the frequency of changes at each source, the load on the network and sources, the request scheduling policy at the sources. The problem of maintaining freshness has analogies with the well studied problem of memory coherency in distributed shared memory architectures, where a similar issue of propagating updates exists [1].

The main contributions of this work are a model of coherence (called slacker coherence) addressing the freshness problem in the context of current pull-based protocols. We carry out an assessment of various techniques for estimating update rates (used to drive the coherence model). Finally, we look at the role of the scheduling algorithm at the source and its impact on our coherence model performance.

2 Related Work

In [5], a general framework is described that supports efficient data structure sharing with client-controlled coherence for interactive and distributed client-server applications. The runtime interface enables clients to cache relevant shared data locally, resulting in faster (up to an order of magnitude) response times to interactive queries. The framework allows the user to control the degree of coherency of the cached copy. Andrade et. al. [1] present middleware approaches for optimizing execution of multiple queries through data reuse with user-defined data structures and operations on data. The middleware implements active semantic caching, query scheduling, and multi-threaded execution on clusters of SMP machines. However, that work does not address the issues pertinent to maintaining the coherence between cached data items and data sources.

Shah et.al [7] propose techniques for delivery of time varying data from data sources to a set of cooperating repositories. They focus on strategies for cooperation among repositories for pushing the data updates to reduce communication and computation overheads. They show that increasing the amount of cooperation does not always result in better performance. Akamai (http://www.akamai.com) is a content distribution system that consists of geographically distributed edge servers that are *pushed* updates from the actual web server. The DNS mechanism redirects the request for a page to the nearest and least loaded Akamai edge server. The edge server not only servers static web content but also serves dynamic content like stock quotes or weather status.

Röhm et.al. [6] address the tradeoff between data freshness and optimization of query evaluation. They target a cluster of databases, and propose a *coordination* middleware that is designed for scheduling queries and coordinating routing of data updates to maximize the throughput of queries. They propose of protocol called *Freshness Aware Scheduling* and show that this protocol performs much better than synchronous replication approaches. Ninan et.al [4] present an approach for maintaining cache consistency in multiple proxy environments. They propose a *cooperative consistency* model and *cooperative leases* mechanism to support it. Their work targets content distribution networks where servers push data to proxies, which sit between servers and clients. Deolasee et.al [3] examine methods that combine pull- and push-based models. They present approaches that adaptively choose between pull- and push-based models or simultaneously apply both models. In the latter case, the proxy mainly is responsible to pull updated data from the server, while the server may push additional updates. Chawathe and Garcia-Molina [2] address methods for detecting updates in data. They present algorithms for comparing data represented in tree structures. Zhuge et.al [10] present algorithms for incremental maintenance of views at a data warehouse when the view spans multiple data sources. The goal is to update the views incrementally while ensuring that the data presented to client queries is consistent and the impact of view maintenance costs on query execution is minimized. Their approach involves the use of *monitors* at the data sources. A monitor identifies updates of interest and notifies the corresponding data warehouse.

Our work has similarities to these projects in that we look at mechanisms for maintaining up-to-date views and data caches. However, we investigate methods for pull-based environments where data warehouses have to poll the data sources for updates. We also examine the effects of data scheduling at the data sources.

3 Architecture Overview

This paper targets a three-tier architecture that consists of clients, ad-hoc data warehouses, and data sources. In this section we briefly describe each of these components.

Clients form the first tier of the architecture. A client or a group of clients can register datasets from multiple data sources, with a particular ad-hoc data warehouse, and cre-

[1]Due to this analogy, we will use the term "coherence" to denote both the freshness problem and the specific set of techniques we use to solve it.

ate one or more views assembled from one or more of these datasets. A view effectively corresponds to an assembly of queries, each of which defines a subset of attributes from the registered datasets and a set of operations on these attributes. For example, a user may specify a spatial and temporal range in a set of medical images corresponding to data from multiple imaging modalities and a set of statistical methods that can be executed on these datasets to detect and extract features. Once this view is defined, the client can further perform such operations as visualization of the features.

A *data source* is the location where input datasets are stored. Data sources respond to queries from and serve data to ad-hoc data warehouses. A data source can be a remote file system, a web service, or a database server. For any type of data source, generating a response to a request will take some time, consuming some amount of local resources. When multiple requests are received, these request should be ordered based on a scheduling mechanism and served in this priority order. Response times seen by the clients of a data source (the ad-hoc data warehouses are the clients) are affected by the nature of the scheduling policy used. We look at different scheduling policies in the next section.

An *ad-hoc data warehouse* is the middle tier component and behaves as an intermediate server between clients and data sources. Any interaction of clients with data sources is carried out through ad-hoc data warehouses. Having an ad-hoc data warehouse relieves clients from maintaining views. In collaborative environments, a group of clients likely create the same or overlapping sets of views. Without a data warehouse, individual clients should maintain data from multiple data sources and construct views spanning data across these sources. If clients directly interact with a data source, the load of the data source will increase as the number of clients increases. By consolidating views at the data warehouse, data duplication is minimized and the load seen by data sources is reduced. Moreover, if clients and data sources are separated by a wide-area network, network demands and the latency incurred between a client and a data source may be very high. An ad-hoc data warehouse, on the other hand, can be instantiated near a group of collaborating clients. As a result, the volume of data transferred across wide-area networks is decreased, resulting in less network overheads.

A data warehouse should allow clients to create views that can span datasets across multiple data sources. In order to serve the clients that use these views, it should retrieve the data subsets that are required by the views from the data sources and store them locally. In addition, it should be responsible for maintaining the quality of the data to suit client requirements. The data warehouse should keep copies of data that are coherent with the data at the data sources, as the data at each site can be asynchronously updated by external sources.

There are two basic ways by which a data warehouse can synchronize with a data source. In the push model, when a data item is updated at the data source, it is *pushed* by the source to the data warehouse. The push model requires that data sources support registration of triggers for multiple views and of data warehouses maintaining those views, and implement mechanism for pushing the data. In the pull model, it is the responsibility of the data warehouse to poll the data sources for updates and retrieve the data when there is an update. It is a natural characteristic of a pull-based model that some of the updates at the source will be missed. In order to provide high quality views to clients, a data warehouse should be designed to minimize the number of missed updates and maximize the freshness of data comprising the view, wherein freshness is defined as the fraction of time the view at the warehouse is in sync with the data source Although maximizing the freshness of data is in many cases closely related to minimizing the number of updates missed, different applications may place different importance to these two criteria. For example, Grid environments are inherently composed of shared resources, and the availability of the resources vary over time. Grid services that monitor Grid resources should provide information as up-to-date as possible. In order for applications to make best use of the resource availability data maintained by the service, the freshness of the cached data should be high. On the other hand, data warehouses that monitor stock markets, the main aim of the warehouse should be to make sure that all price changes at the source are captured. The main objective in this case is to minimize the number of updates that are not propagated to the data warehouse.

4 Coherence Protocol and Scheduling Policies

In order to achieve low response times for client requests and scalable performance, an ad-hoc data warehouse should locally cache the required subsets of data from data sources. The data warehouse should also implement a consistency model so as to serve up-to-date data products. Given the high latencies and variable network performance of typical distributed environments on the Internet, the use of memory consistency models similar to those provided by hardware- and software-based shared memory systems is not always the most efficient alternative. In our context, ad-hoc data warehouses can often accept a significantly more relaxed—and hence less costly—coherence and consistency model. Basically in such domains, a stale view may be acceptable up to a point. The level of tolerance here may be defined in terms of time units by which the local cached data is out of date, or the percentage out-of-datedness of the local data copy with respect to the data source's copy of the data.

Without a push-based model, it is the responsibility of the data warehouse to poll the data sources for updates to the data of interest. In order to efficiently carry out this task, a polling rate should be determined for each of the views maintained by the data warehouse. If a view spans multiple data sources, each of the data sources may need to be polled. A simple approach would be to query the data source(s) at regular intervals.However, there are several problems with this strategy. First, although overly frequent polls will ensure that the tolerance metric is met, it results in high communication overhead and bandwidth utilization. Under-polling, on the other hand, results in the data warehouse view being always out of date. Second, the frequency of updates at the data source should be known a priori to the operation of the data warehouse. In general, this assumption will hardly hold unless the applications that update the data at the data source exhibit a well-defined behavior or the data source operates under a controlled environment. Second, the update rate will probably vary over time. As a result, the polling rate should be higher than the expected maximum update frequency. If the polling rate is matched to the maximum update frequency, resources in the environment are wasted because of unnecessary polls. To address these problems, we propose a slacker coherence protocol described next.

4.1 Slacker Coherence Protocol: To Poll Or Not To Poll

The main premise behind the slacker coherence model is to adaptively determine the ideal polling rate based on the frequency of updates at the data source side for a given view while meeting the ad-hoc data warehouse's tolerance metric for stale views. The basic idea is to decrease the rate of polling for a view and of a data source, if the data of interest is not being updated. In order to meet this objective we first need to reliably estimate the update rate at the data source. To do so we maintain a window W of information about previous poll requests and update rate information from the data source. The warehouse then uses this time series information to estimate the likely time of the next update.

We consider three different levels at which a data source can maintain information about the update timestamps that it returns to the data warehouse upon a polling request.

No Information: The data source returns the updated data only. In this case, the window based estimation schemes fail. In order to estimate the update rate, we use an *Adaptive Polling* strategy, wherein the data warehouse polls the data source after a time interval t. If the poll results in a value that is significantly different from the value that is currently maintained by the warehouse, the warehouse decreases the time to next poll to $t * (1 - d)$, where d is the damping factor and hence increase the polling rate. However if the poll did not result in any significant change in value, the warehouse increases the time to next poll to $t * (1 + d)$ and hence decrease the polling rate.

Last Update: The data source returns the updated data and the time stamp of the last update on that data. The data warehouse can maintain a time series of the update times using the last update timestamps returned by the source. However such a series is likely to be less accurate as it does not capture the dynamics of update at the source fully. This kind of scenario is likely to emerge in the case of data sources like web servers that use HTTP to serve data requests.

All Updates: The data source returns the updated data and a history of time stamps of all intervening updates since the last poll by the data warehouse to the data source on that particular data.The data warehouse has the most accurate time series information for estimating the update rate at the data source. This type of estimation works when the data sources is capable of sending the history of updates to the view along with the view itself, which is possible when the data source is a database server as it typically maintains logs of update timestamps.

To estimate the update rate, the window of information, W, contains information on the previous K updates. This history information allows us to derive a mean update rate, which is used to determine the time to the next poll. The larger the value of K is, the less susceptible the protocol is to noise. However, in this case the hysteresis (delay to adapt to a new update rate) time will be longer. Hence the optimal window size will be dependent on the update pattern of the view at the data source under consideration. An important point to note is that currently the update estimation protocol ignores the vagaries of the scheduling policy at the data source.

4.2 Scheduling Policies at the Data Source

When a data source receives multiple requests from data warehouses, it needs to schedule these requests to generate a response for each of them. Therefore, in addition to the coherence model the influence of the scheduling policy at the data sources on the coherence model bears closer inspection. In particular, we look at the following three scheduling policies. These policies differ from one another in that each requires the data source to maintain different levels of knowledge of the updates and the environment. 1) **First In First Out (FIFO)**. The requests are served in the order they are received. This is one of the most basic scheduling policies commonly available in most data servers. 2) **Least Recently Requested (LRR)**. The idea behind this policy is to reward the data warehouses that take advantage of the slacker coherence model and do not poll overly often. The scheduler prioritizes requests from data warehouses that have not recently polled the data source. To im-

plement this strategy the data source needs to keep track of the time of the last poll by each ad-hoc data warehouse. 3) **Most Frequently Changed (MFC)**. The idea here is to prioritize requests for data items that are most frequently updated. The advantage over the LRR scheme is that the data source does not have to maintain state information about all its ad-hoc data warehouses and the last time they polled for an update. Moreover, this approach tends to prioritize data that are updated more frequently which lends itself to the overall objective of keeping most of the views up-to date.

5 Experimental Results

Experimental Setup We present a preliminary experimental evaluation of the slacker coherence protocol performance. We measured how different aspects of the design of a system employing slacker coherence affect its performance: we looked at polling strategies, query scheduling policies, tolerance to data staleness, scalability with respect to number of warehouses, effect of window size, and of the damping factor (for the adaptive polling). We carried out experiments on a testbed consisting of two clusters connected through the OSU campus network. The Data sources ran on nodes that were Pentium III dual processor nodes running Linux 2.4 with 1GB of memory and interconnected using Fast Ethernet. The Data Warehouses ran on nodes that were Pentium III uni-processor nodes running Linux 2.4 with 1 GB of memory and interconnected using Fast Ethernet. For all experiments unless mentioned otherwise, the data sources run on one cluster and the data warehouses on the other.

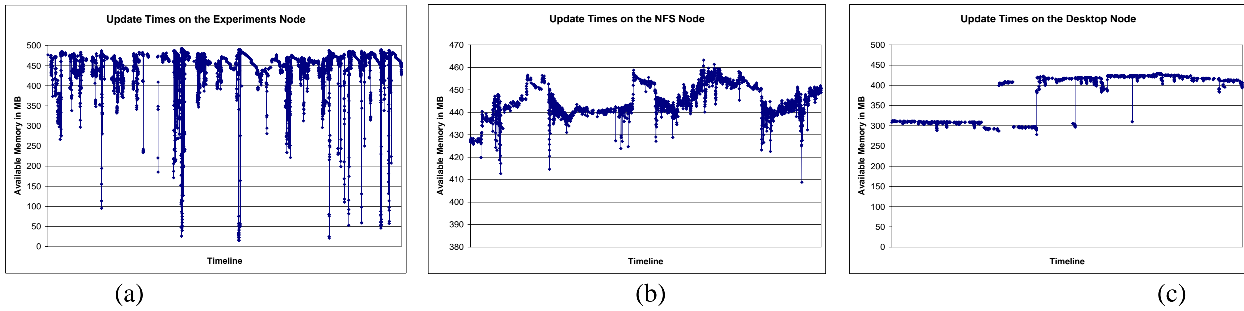
The experiments were carried out using 2 data sources and 12 data warehouses. The data sources maintain three base tables that have different update rates associated with them. The base tables are modeled as flat files. The first set of experiments used a synthetic trace of updates to the three tables, generated according to a Gaussian distribution of update intervals. The second set of experiments used a set of three real traces of update patterns. The traces represented time series measurements of available memory on machines with different usage patterns; one was a NFS Server, another an Experimental Node running data intensive applications, and the last was a desktop PC. Figure 1 shows the update time points and the corresponding amount of change in available memory. For our experiments we use only the series of time points and we ignore the value of available memory; each trace is associated to a table, and a time point in the trace represents the instant of time when an update to the corresponding tables occurs.

Each Data Warehouse maintains a view built on one base table from each data source. Each data warehouse sends a poll request to the corresponding data source at the time it estimates the next update will occur. Each poll consists of a request-response pair and hence the warehouse waits for a reply from the data source before sending the next poll request to any other data source.

Polling Strategies The first set of experiments shown in Figure 2 examines the effect of the granularity of information on updates that can be maintained by a data source. We present results for both the synthetic and the real trace of updates at the data source. The scheduling at the data sources is FIFO for all experiments in this section unless otherwise mentioned. The Push-based polling case, wherein the warehouse polls only when it is notified of an update by the data source, gives the best performance both the terms of number of polls and the freshness of data. This serves as a baseline against which to compare performance of other schemes. The all-updates case has the most accurate time series information and hence gives a better performance value compared to the last updates case. Doubling the rate of polling helps in decreasing the number of updates missed. The adaptive polling strategy gives a comparable freshness value though it was observed that the total number of polls made were higher than the other strategies. The standard deviation based adjustment gives a more than proportional increase in freshness values.

An interesting result is that the slacker coherence protocol seems to be quite robust with respect to the distribution of updates. Figure 2 shows that the behavior of the different polling strategies is substantially the same for update patterns as diverse as the synthetic and real traces. In the remainder of the paper we will present the results obtained with the memory update based traces alone due to lack of space. The results for the Gaussian distribution can be found in [8].

Tolerance In this experiment we vary the tolerance on the maximum allowed number of outdated records composing a view. A view at a data warehouse is considered *fresh* if the number of out-of-date records is below a user-defined fraction. The warehouse can choose to maintain data that is not fully synchronized with the data source, therefore allowing a predetermined percentage of records to be out of date. For example a warehouse that chooses to maintain data with a 10% outdateness tolerance will poll only when its estimate of outdated record becomes larger than 10%. Results for the experiment are shown in Figure 3. These experiments were done with FIFO scheduling at the data source. The figure shows the freshness of views for different levels of tolerance. As expected, the higher the tolerance level, the higher the freshness achieved by the view.The slacker coherence polling ensures that the number of polls to the data source decreases accordingly.



(a) (b) (c)

Figure 1. Memory Update trace (a) Experiments Node (b) NFS Server, (c) Desktop Node.

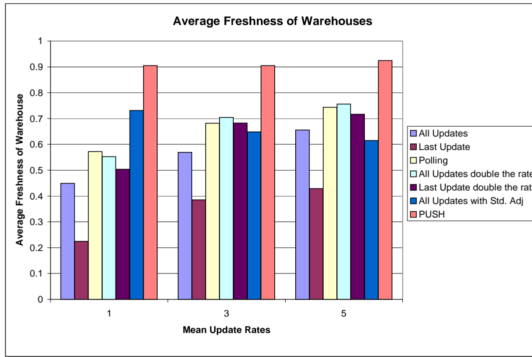
Scheduling Policies In these experiments, we look at how query scheduling policies on the data source affects the freshness of views at the data warehouses. Figure 4(a) shows the effect of the different scheduling policies when the size of updates at the data source is a constant value of 700KB. We assume each query results into a whole table (ranging in size from 10MB to 20MB) being sent to the warehouse. Figure 4(b) shows the effect of the different policies when the size of updates is not a constant. We used another real trace to decide the size of updates: the amount of free CPU measured using the Network Weather Service [9]. The amount of free CPU cycles, derived from a trace that was collected in synchrony with the memory traces, is translated into an update size ranging from 1KB to 700KB (mean update size is 350KB). Since we are using a CPU trace, the size of the query result, and thus indirectly the query response time, reflects the usage pattern of the source. As shown in Figure 4(a), the larger growth rate of the view, which results in greater query overheads, increases the importance of the scheduling scheme. The MFC scheduling policy is very effective in differentiating service between the more frequently changing Experiment Node trace and the less frequently changing Desktop Node trace.

The LRR policy allows for a more fair scheduling, which works well for the less frequently changing views but penalizes the more frequently changing views. This in some sense is in line with the slacker coherence policy. The number of polls made by the warehouses for these views follows the same pattern as the freshness values: the number of polls made by each view is proportional to the frequency of change at the data source (smaller for the Desktop than for the Experimental Node trace). The data source contributes to the diversity in polling rates through the differentiated treatment of queries. Figure 4(b) shows a similar trend, but the effect of the scheduling policy on the experimental node trace also reflects the way the size of the view varies. Since the average update size is smaller than in the previous experiment, the influence of scheduling on the freshness of views maintained by the warehouses is smaller.

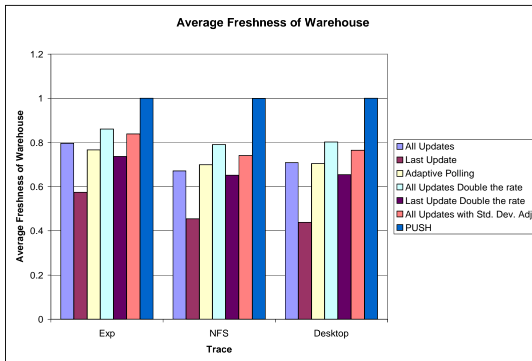
Increasing number of Warehouses The effect of increasing number of warehouses is shown in Figure 5. The experiment consists of a single data source with the same base tables as described above. The number of warehouses that request a view from the data source is increased and the objective function that is being studied here is the overall average freshness of data across all the data warehouses. Increasing the number of warehouses does have a negative impact on the overall system performance, as expected. However the freshness degrades quite gently, showing that the slacker coherence protocol improves the overall system scalability. Another interesting result is the decrease in the number of polls as the number of polling warehouses is increased. This decrease is the result of the data source increased per-poll response time due to congestion.

Window Size Figure 6(a) shows the effect on performance of the size of the time series window maintained by the warehouse. We observed that there is an optimal window size that can effectively capture the dynamics of the update pattern. Nodes which exhibit a larger non-periodic rate of change will require a smaller window size for better response times, while nodes which exhibit a smooth fluctuation or infrequent changes will require a larger window size in order to capture this behavior. Figure 6(a) illustrates this behavior for the three categories of traces analyzed. The trace of experimental nodes requires a window size of 10 and Desktop nodes require a window size of 25 for optimal performance, which conforms to their usage patterns.

Adaptive Polling – damping rate The Adaptive Polling Scheme uses the least amount of information from the data source to decide the polling rate. The main parameter in this scheme is the damping factor, that affects how quickly the polling rate is increased or decreased. The higher the damping rate, the faster is the response to change and higher the oscillation. Lower damping rates result in less oscillations and are more suited for smoother trends but do not perform well when there are sudden shifts in the usage pattern. The graphs in Figure 6(b) illustrate the effect of varying damping factors on freshness of data at the warehouse. The



(a)



(b)

Figure 2. The effect of the granularity of information maintained by a data source. (a) Synthetic trace of updates at data source (Gaussian distribution), (b) Real trace of updates at data source (memory usage).

heavily loaded experimental workstation node has a spiky behavior and hence has an optimal damping rate of 50%. However the NFS server and desktop machines, which exhibit more slow fluctuations, have an optimal damping rate of 25%.

6 Conclusions

In this paper we examined a slacker coherence model to address the freshness problem in the context of pull-based models. Our results show that the proposed approach reduces the number of polls to data sources, while maintaining relatively up-to-date views. We also observe that the level of cooperation from the data source in terms of the amount of information about updates and scheduling policies affects the performance of the slacker coherence protocol.

References

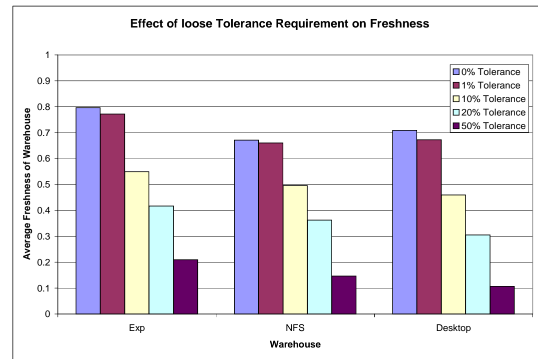


Figure 3. The effect of the tolerance on freshness of views at the data warehouses

[1] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. In *Proceedings of the Fifth International Parallel Processing Symposium & Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, April 2002.

[2] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD Conference*, pages 26–37, 1997.

[3] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *World Wide Web*, pages 265–274, 2001.

[4] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable consistency maintenance for content distribution networks. Technical report, University of Massachusetts, Amherst, 2001.

[5] Srinivasan Parthasarathy and S. Dwarkadas. Shared state for distributed interactive data mining applications. *The International Journal on Distributed and Parallel Databases*, March 2002.

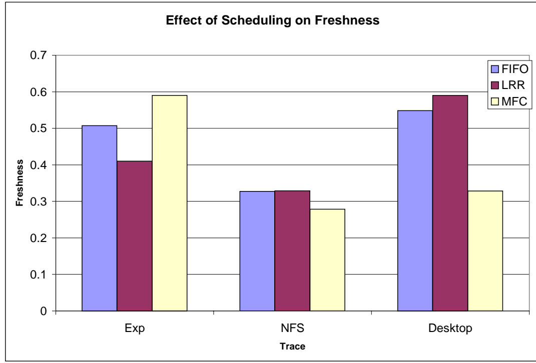
[6] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components, citeseer.nj.nec.com/529123.html.

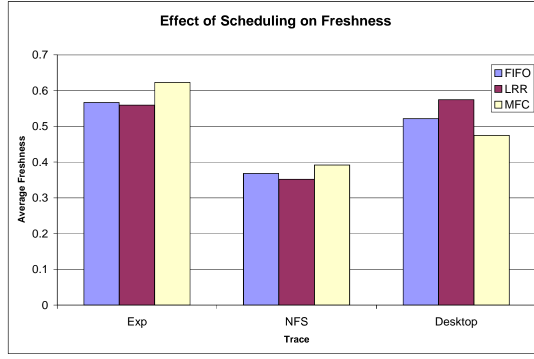
[7] Shetal Shah, Krithi Ramamritham, and Prashant Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, August 2002.

[8] R. Sundaresan, T. Kurc, M. Lauria, S. Parthasarathy, and J. Saltz. A slacker coherence protocol for pull-based monitoring of on-line data sources. Technical Report OSU-CISRC-2/03-TR08, Department of Computer and Information Science, The Ohio State University, February 2003.

[9] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

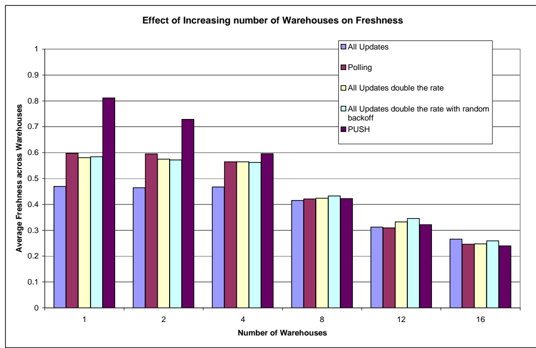
[10] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Journal of Distributed and Parallel Databases*, 6(1):7–40, 1998.

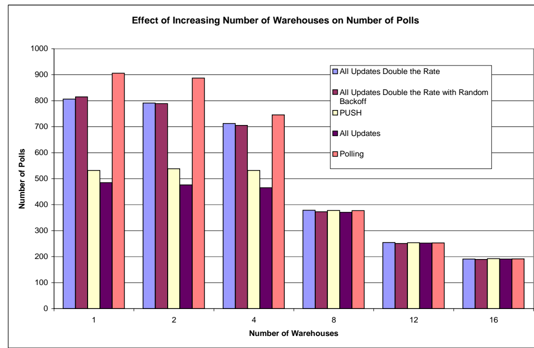




(a) (b)

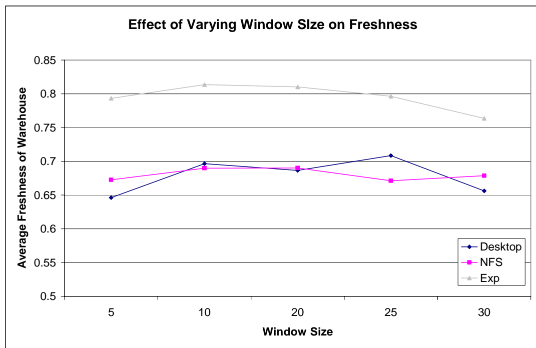
Figure 4. The effect of the scheduling of queries at a data source. Percent freshness of views at the data warehouses (a) For constant update size, (b) variable cpu trace based update size.

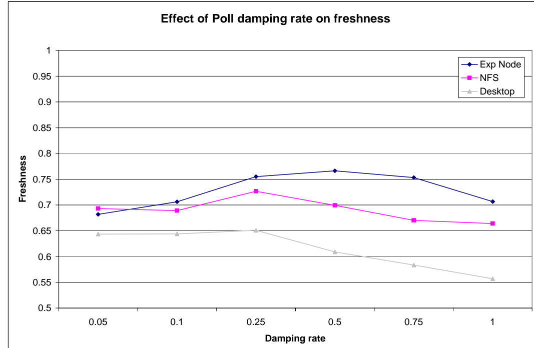




(a) (b)

Figure 5. The effect of varying the number of data warehouses. (a) The freshness of views. (b) the number of polls.





(a) (b)

Figure 6. Average Freshness of views (a) varying window size and (b) adaptive polling – varying damping rates.