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

There is an increasing number of online applications that operate on data from disparate, and often wide-spread, data sources. 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, 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 current pull-based protocols, and examine the role of the request scheduling algorithm at the source and its impact on our coherence model performance.

1 Introduction

One of the consequences of the explosive growth of the Internet and in particular the world wide web (WWW), is the explosion of distributed online applications that operate on disparate, and often wide-spread, data sources. Data and transaction based commercial applications such as on-line auctions, financial markets data, satellite data, airline reservations, medical point-of-care data are figuring prominently among Internet-based information providers. Other examples of information delivered through the Internet (often via WWW) include scientific data in fields such as biology, medical/health sciences, genomics/proteomics, astrophysics, geophysics, where an increasingly large share of publicly funded projects are aimed at the production and update of extensive data collections to be made available to the research community. Finally, experimental distributed applications such as grid meta data catalogs, are also taking advantage of the decentralized nature of the WWW and Internet protocols.

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. Monitoring and cross-correlation of data on infectious disease cases, environmental contamination or accident reports could be used for homeland security purposes.

On another front, there is an increasing trend towards technologies to support the information, data sharing, and data analysis services needs of often ad-hoc groups of institutions and researchers. Such collaborative environments often involve studies that have the need to pool and analyze data from disparate sources. Composite information systems that will support such studies have great potential of making a significant impact on research outcomes and their application.

Federated database technology has been developed to provide unified access to distributed and diverse sets of data. These systems provide virtualizations of data through heterogeneity transparency and distribution transparency. 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 charac-

[*]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).

teristics of the services should be. As a result, a client has to be able to interact with services with different capabilities. In a distributed environment, a client should be able to formulate queries that can span large data sets at multiple data sources (exposed as Web services or Grid services). Moreover, the client should be able to monitor the data sources for updates to the data of interest.

In a client-server 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 and 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 between 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 to make it immediately usable with existing web-based data sources. In a sense, our ad-hoc data warehouse concept fills a functionality gap left by current protocols. The purpose of our study is to identify the most effective techniques of filling the gap.

Maintaining the freshness of views when using the 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, the insight into the role of the scheduling algorithm at the source and its impact on our coherence model performance, and finally an assessment of various techniques for estimating update rates (used to drive the coherence model).

[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.

2 Related Work

In [6], 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. It also supports relaxed coherence models including temporal, diff-based, and delta coherence.

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. The active semantic caching allows user-defined data structures for intermediate and final results for a query to be cached and reused by other queries. However, that work does not address the issues pertinent to maintaining the coherence between cached data items and data sources.

Shah et.al [8] 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. [7] address the tradeoff between data freshness and optimization of query evaluation for OLAP applications. 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 OLAP queries. They propose of protocol called *Freshness Aware Scheduling* and show that this protocol performs much better than synchronous replication approaches.

Ninan et.al [5] 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 [9] 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 create 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 and will consume 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. Hence, the behavior and performance of the ad-hoc data warehouses monitoring the data source will likely be dependent on the scheduling policy at the data source. We shall describe two 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 has several performance advantages. First of all, it 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. Second, 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. Third, 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 for the data warehouse 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, for a given view, 1) the number of missed updates and 2) the age of data comprising the view, i.e., the difference between the time of update at the source and the time the data warehouse updates its local copy. 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 age of the cached data should be low. 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. The most relaxed of these, release consistency [4], guarantees a coherent view of *all* shared data at synchronization points, resulting in significant amounts of communication.

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. In other words, such applications may tolerate stale data based on a temporal or change-based criterion, thereby allowing a reduction in communication overhead to improve overall efficiency. Performance can be further improved by allowing each ad-hoc data warehouse to specify the coherence model required for correct operation.

While several existing systems offer such a view, almost all of them rely on some form of server-based (data-source based) push model. It is our contention that in many situations one will not have control over whether the push-based model is actually feasible or available. 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 can be polled at different rates to reduce network traffic. Several factors such as client-assigned priorities, the number of clients served by a given view, and the cost of processing requests to a view can be taken into account in determining the frequency of polling operations. In addition to these client-oriented factors, the update rate at the data source must be considered to meet the tolerance metric.

A simple approach would be to query the data source(s) at regular intervals. If the polling rate is higher than the data update rate at a data source, this strategy guarantees that the local copy of the data of interest will always be up-to-date. 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.

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

Level 1. The data source returns the updated data only.

In this case, the data source does not need to maintain any additional information. However, the data warehouse has to use the local polling logs to estimate the age of the locally cached data as well as the update rate at the data source.

Level 2. The data source returns the updated data and the time stamp of the last update on that data. Unlike the first level, the data warehouse can now calculate the age of the local data more accurately. However, the data source should be designed to maintain the time stamp of the last update to the data items. The last-update timestamp information is available from data sources like web servers that use HTTP to server data requests.

Level 3. 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. At this level, the data warehouse obtains all the information that can be used to do a better job in 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. However, this is achieved at the expense of increased network resource consumption, as more information is sent from the data source to the data warehouse, resulting in higher network overhead.

To estimate the update rate, the window of information, W, contains information on the previous K updates. This historical 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. Once we have an estimate of the update rate we can determine how often to poll for a view given the tolerance constraints at the data warehouse. If the polling rate is too fast or too slow the feedback via the update rate estimation will enable the system to slow down or speed up the polling adaptively. 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 implement 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

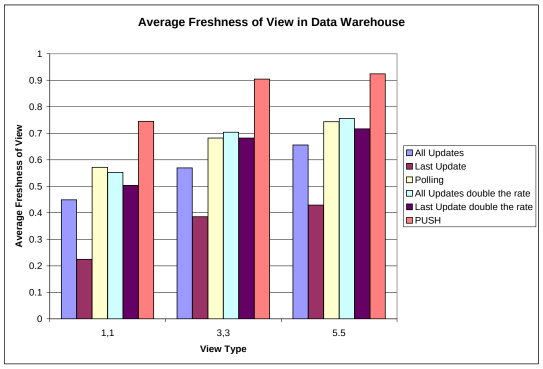
We present a preliminary experimental evaluation of the slacker coherence protocol and the scheduling policies presented in this paper. We carried out experiments on a testbed consisting of two clusters connected through the OSU campus network. The Data sources ran on nodes that were Pentium III dual 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 done using 2 data sources and 6 data warehouses. Each data source maintains three base tables, and the update rate follows a Gaussian distribution with a mean of 1, 3, and 5 seconds respectively. The base tables are modelled as flat files. Each Data warehouse maintains a view built on one base table from each of the two data sources. Each data warehouse sends a poll request to the corresponding data source at the time it estimates the next update will occurr. 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. We choose the request-response model because it is the polling mechanism supported by typical data sources like HTTP-based web servers and database systems. The query processing time at the data source is fixed at 200ms. The total observation time for all experiments was 600 seconds. In these experiments, we measure the freshness of the view at the data warehouse and the total number of polls to a data source. Freshness is defined as the fraction of time during which the view is updated with respect to the the data at the source. Since experiments were conducted using a small number of warehouses and data sources, source side resources such as the request queue length were reduced proportionally to simulate a situation when the server is heavily loaded.

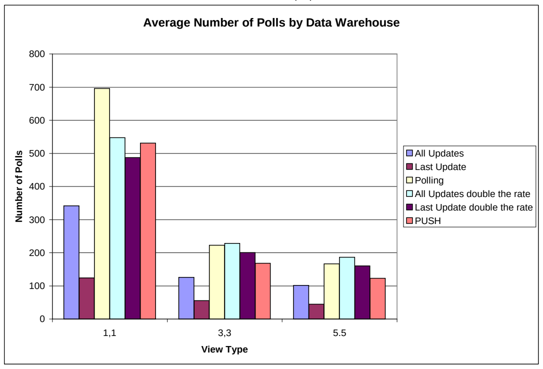
The first set of experiments shown in Figure 1 examines the effect of the granularity of information on updates that can be maintained by a data source. We look at five cases: 1) The data source maintains the timestamps for all updates to each table (**All updates**). 2) The data source keeps the timestamp of the last update to each table. In these two cases, the data warehouse estimates the rate of updates using the information maintained by the data source (**Last update**). 3) No update information is stored at the data source. The data warehouse estimates the update rate based on the results of the poll queries. If the result of a poll query is "no update", the data warehouse decreases the current polling rate by 10%. On the other hand, if the poll query returns an updated result, the current polling rate is increased by 10% (**Polling**). 4) The data source maintains information for all updates. But, unlike case 1, the data warehouse polls the data source at twice the estimated update rate (**All updates double the rate**). 5) The data warehouse uses the time stamp of the last update, but as in case 4 it polls the data source at double the estimated update rate (**Last update double the rate**). We also implemented a simple push model (**PUSH**). In this model, the data source notifies the data warehouse when there is an update to the table comprising the view. The data warehouse then sends a poll query to the data source to receive the updated information. In the figures, *View Type* axis denotes the view maintained by a warehouse. A view (X, Y) is created from two tables, the first of which is updated at a rate of X seconds, while the other is updated at a rate of Y seconds. The experiments were done using FIFO scheduling at the data sources.

As is seen in Figure 1(a), case 1 does better than case 2 in terms of the freshness of views across all views, because we have more information on the update rate. For views (3,3) and (5,5), case 4 does better than case 3 and hence the slacker polling model is better than polling. However, for view (1,1) adaptive polling does better than case 4 in terms of freshness of data. This is because the view is changing at a fast rate (1 sec) and hence the chances of error in estimation is higher. Figure 1(b) compares the number of polls done for each view type.

In the second set of experiments, we look at the effect of scheduling policy on the freshness of views at data warehouses and the number of polls. Figure 2 show the average freshness of views with different scheduling policies at the data source. For view (1,1), MFC does the best because MFC prioritizes requests for the base table that are changing frequently. It should also be observed that view (1,1) does not do well for the LRR case since LRR prioritizes



(a)

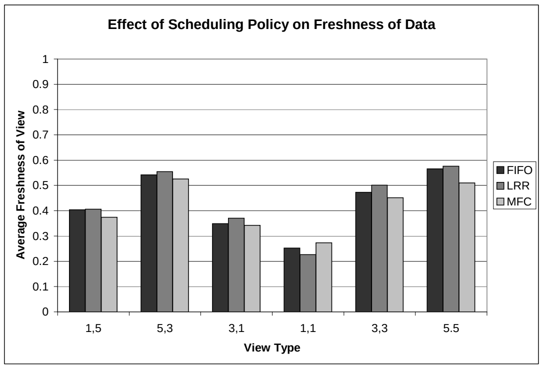


(b)

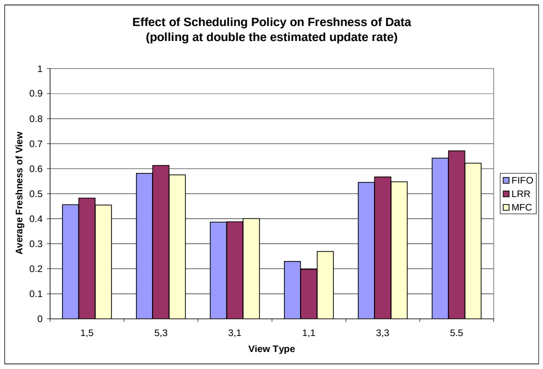
Figure 1. The effect of the granularity of information maintained by a data source. (a) percent freshness of views at the data warehouses, (b) the number of polls to the data source.

requests from warehouses that are less frequently polling. This trend is clearly seen for views (3,3) and (5,5).

In the third set of experiments, we vary the tolerance on the number of outdated records composing a view. That is, 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 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. The higher the tolerance level is, the higher the freshness of the view, as expected In our experimental conditions, none of the schemes is able to meet the required tolerance. The slacker coherence polling ensures that the number of polls



(a)



(b)

Figure 2. The effect of the scheduling of queries at a data source. Percent freshness of views at the data warehouses (a) when polled at the estimated update rate, (b) when polled at double the estimated update rate.

to the data source decreases accordingly.

Figure 4 shows the effect of slackware coherence window size on performance. This experiment was done with FIFO scheduling at the data source. The warehouse maintains a window that maintains the history information about a finite number of past update times and uses this history information to calculate a mean of the update rate at the data source. We observe that increasing the window size reduces the number of polls, but also decreases the freshness of data. However, increasing the window size after an optimal point does not improve the overall freshness and does not reduce the number of polls. With small window size, fluctuations at the update rates at the data sources affect the estimated update rate. As the window size is increased, the estimate is computed over more data points, hence update rate fluctuations are smoothed out.

The effect of increasing number of warehouses is shown in Figure 5. This experiment was done with FIFO scheduling policy at the data source. 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 does not decrease sharply. This shows that the slacker coherence protocol helps the overall system scale when the number of data warehouses is increased.

6 Conclusions

In this paper we examined a slacker coherence model to address the freshness problem in the context of pull-based models. Our preliminary 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

[1] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. In *IPDPS 2002*.

[2] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, 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 *WWW*, 2001.

[4] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA 1990*, May 1990.

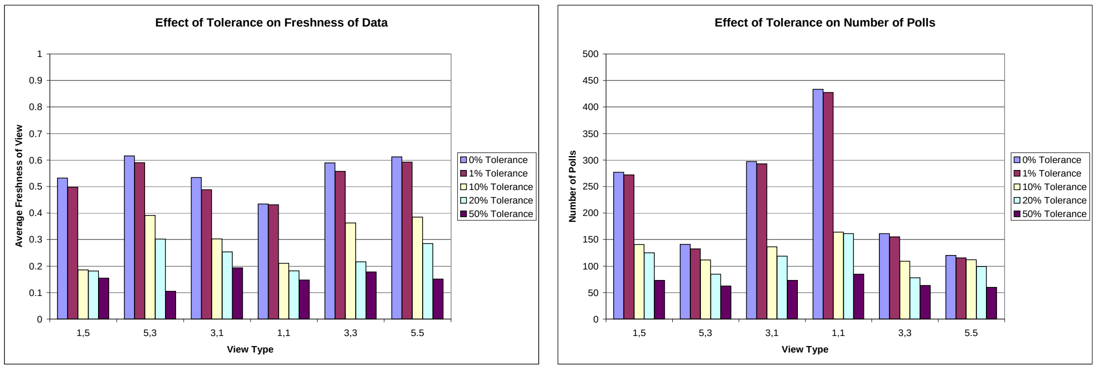
[5] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Scalable consistency maintenance for content distribution networks. Technical report, University of Massachusetts, Amherst, 2001.

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

[7] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components.

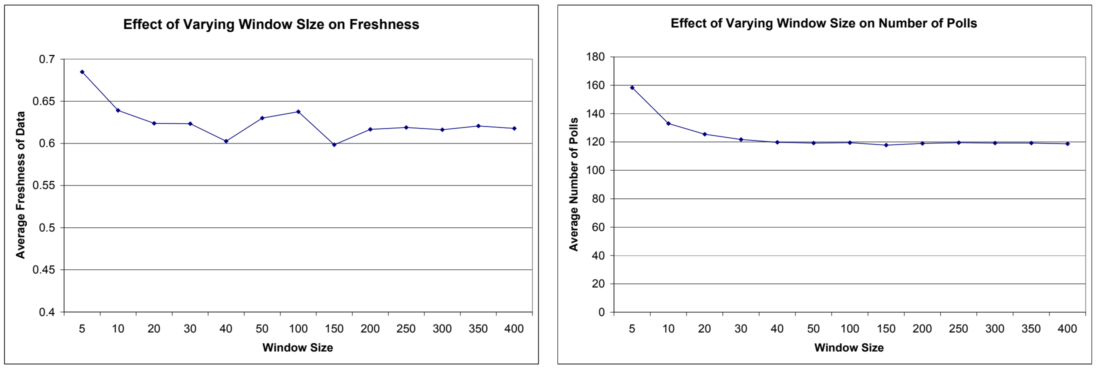
[8] Shetal Shah, Krithi Ramamritham, and Prashant Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB 2002*, August 2002.

[9] 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.



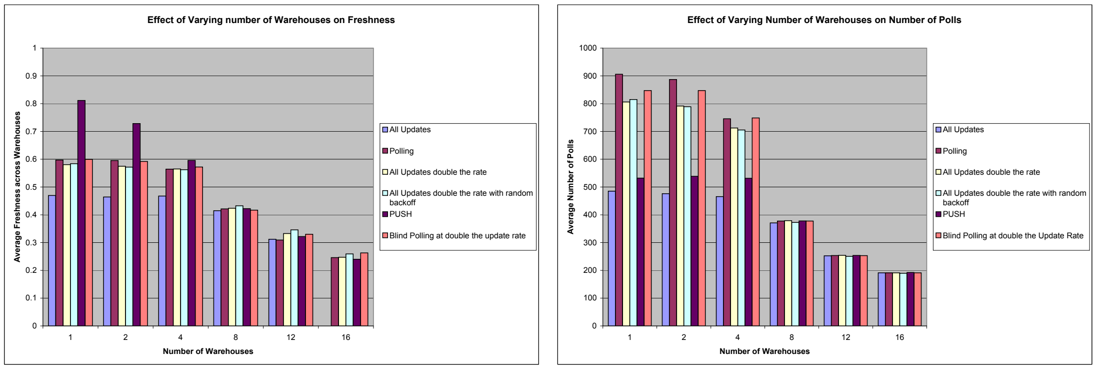
(a) (b)

Figure 3. The effect of the tolerance. (a) percent freshness of views at the data warehouses, (b) the number of polls to the data source.



(a) (b)

Figure 4. The effect of the window size on (a) average freshness of views and (b) the number of polls.



(a) (b)

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