Shared State for Distributed Interactive Data Mining Applications *

Srinivasan Parthasarathy
Computer and Information Science
Ohio State University,
Columbus, OH 43235
srini@cis.ohio-state.edu

Sandhya Dwarkadas
Computer Science
University of Rochester
Rochester, NY-14627
sandhya@cs.rochester.edu

October 3, 2001

Abstract

Distributed data mining applications involving user interaction are now feasible due to advances in processor speed and network bandwidth. These applications are traditionally implemented using ad-hoc communication protocols, which are often either cumbersome or inefficient. This paper presents and evaluates a system for sharing state among such interactive distributed data mining applications, developed with the goal of providing both ease of programming and efficiency. Our system, called InterAct, supports data sharing efficiently by allowing caching, by communicating only the modified data, and by allowing relaxed coherence requirement specification for reduced communication overhead, as well as placement of data for improved locality, on a per client and per data structure basis. Additionally, our system supports the ability to supply clients with consistent copies of shared data even while the data is being modified.

We evaluate the performance of the system on a set of data mining applications that perform queries on data structures that summarize information from the databases of interest. We demonstrate that providing a runtime system such as InterAct results in a 10-30 fold improvement in execution time due to shared data caching, the applications' ability to tolerate stale data (client-controlled coherence), and the ability to off-load some of the computation from the server to the client. Performance is improved without requiring complex communication protocols to be built into the application, since the runtime system uses knowledge about application behavior (encoded by specifying coherence requirements) in order to automatically optimize the resources utilized for communication. We also demonstrate that for our benchmark tests, the quality of the results generated is not significantly deteriorated due to the use of more relaxed coherence protocols.

*This is an expanded version of our conference paper in the First SIAM International Conference on Data Mining [36]. This work was supported in part by NSF grants CDA–9401142, EIA-9972881, CCR–9702466, CCR-9988361, and CCR–9705594; and an external research grant from Compaq.

1 Introduction

The explosive growth in data collection techniques and database technology has resulted in large and dynamically growing datasets at many organizations. For these datasets to be useful, data mining, the process of extracting useful information from such datasets, must be performed. The datasets at these organizations are typically in a remote repository accessible via a local or inter-network. Despite advances in processor and network technology, remote data mining is made difficult by the prohibitive bandwidth requirements imposed by the size of the data involved and the low latency requirements imposed by the interactive nature of data mining. The size of the datasets prohibit transferring the entire data to the remote client(s). In addition, data mining is often an iterative process with the user tweaking the supplied parameters according to domain-specific knowledge. This compounds the problem of increased response times due to network and server delays.

These applications can often be structured so that subsequent requests can operate on relatively small summary data structures [41]. Once the summary structure is computed and communicated to the client, interactions can take place on the client without further communication with the server. Also, the mining processes are often independently deployed and perform very different operations, resulting in different traversals of the shared data. The summary is based on the snapshot of the actual data at any point in time. If the data is dynamically being modified, the summary is likely to change. In this scenario, the client's copy of the summary structure must be kept up-to-date.

The above communication can be accomplished by employing some form of message passing or remote procedure call (RPC) in order to keep the data coherent. However, these techniques are often cumbersome, and can be inefficient. RPC mechanisms work well for "function shipping"—moving the process to the data—but they do not work well for moving data to the process. Using message passing requires users to invent ad-hoc communication and coherence protocols in order to manage data copies. Programming ease concerns suggest the need for an abstraction of shared state that is similar in spirit to distributed shared memory (DSM) semantics. However, even the most relaxed DSM coherence model (release consistency [19, 11]) can result in a prohibitively large amount of communication for the type of environment in which data mining may typically be performed. These applications can often accept a significantly more relaxed—and hence less costly—coherence and consistency model, resulting in excellent performance gains. In fact, as we show in this paper, some tasks might require updates to shared data at regular intervals instead of whenever the data is modified, while others might require updates whenever the data is modified "by a certain amount". In other words, such applications can tolerate stale data based on a temporal or change-based criterion, thereby reducing communication overhead and improving efficiency. Hence, overall system performance can be improved by allowing each client to specify the data shared as well as the coherence model required for its needs.

In this paper, we describe our runtime framework, called *InterAct* [35, 36], developed with such active data mining applications in mind, that allows efficient caching and sharing of data among independently deployed clients and servers. *InterAct* supports data sharing efficiently by communicating only the modified data, and by allowing individual clients to specify relaxed coherence requirements on a per data structure basis. Additional features of the system include dynamic data placement for memory locality (handling the problem of different traversals of the shared data by different processes and made feasible because of the address-independent nature of the system) and the ability to supply clients with consistent copies of shared data even while the data is being modified (made feasible through the use of virtual memory mechanisms to implement coherence and consistency).

The interface is general enough to support a wide range of application domains including the visualization of scientific simulations and the remote tracking of images. We focus here on several applications from the interactive data mining domain and use them to demonstrate the advantages of the *InterAct* system. These applications are structured so that the server is responsible for creating the data structure(s) (storing the summary), mapping them to a *virtual shared dataspace*, and subsequently keeping them up-to-date. The client can then map the data structure(s) from the *virtual shared dataspace* under an appropriate coherence model. We show that executing queries using the appropriate summary structure can improve performance significantly; up to a *23-fold improvement* in query execution times was observed. When the clients cache the summary structure using relaxed coherence models, we also observed *several orders of magnitude reduction in update costs*. Furthermore, for these applications, using such relaxed models does not significantly affect the quality of the results (we observed <2% degradation in result quality).

The rest of this paper is organized as follows. In Section 2, we outline our overall system design goals and compare our system to related work in Section 3. In Section 4, we describe the interface and implementation of *InterAct*, and demonstrate its use through an example application. We evaluate the utility of the system using several applications from the interactive data mining domain, described in Section 5. Experimental results are presented in Section 6. Finally, our conclusions and on-going work are outlined in Section 7.

2 Design Goals

In order to accomplish the goal of efficiently providing shared state in a distributed environment, the runtime system must provide an interface that defines a mechanism to declare shared data that is address space independent and persistent (so that clients can join and leave at any time). In addition to the above minimal requirement for sharing, data mining applications have several properties that can be exploited, and key needs that ideally must be supported.

First, since clients have differing needs in terms of how up-to-date a copy of the data is acceptable, the system must identify, define, and support different relaxed coherence models that may be exploited for application performance. This feature of *client-controlled coherence* is similar to the notion of quasi-caching [4] (see Section 3).

Second, many data mining applications require the capability of obtaining a *consistent* version of a shared data structure at any time, even in the presence of an on-going update to the data. We refer to this feature as *anytime updates*.

Third, many data mining applications traverse these summary structures in an ordered manner. Different clients may have different access patterns depending on the kind of queries processed. This feature requires that the system export programmer-controlled primitives that allows data to be remapped or placed in local memory in a manner that mirrors how the data is likely to be accessed by a given user or client. We refer to this feature as *client-controlled memory placement*.

Fourth, the shared data, although significantly reduced (summary), can still be quite large, so re-sending it on each update can cause significant delays on a busy network. It is therefore important for the runtime system to *identify which parts of the shared data have been modified* since the client's last update. Only the changes need be sent to the client on an update. In Section 4, we describe how our system, *InterAct*, can support the above data mining requirements while providing a general interface for a large class of applications.

3 Related Work

There is a rich body of literature studying the issues in caching and data sharing in many different computing environments. Distributed object-based systems [6, 26, 25, 30, 45, 51, 53, 7, 18], all support the basic requirement of sharing address-independent objects. However, update propagation in such systems, which are typically supported either by invalidate and resend on access or by RMI-style mechanisms, are inefficient (re-sending a large object or a log of operations (RMI)) and often infeasible (especially if the methods require data available only on the server side) for data mining applications. Distributed shared memory systems [3, 5, 11, 29, 46, 44, 22, 24, 55] all support transparent sharing of data amongst remote processes, with efficient update propagation, but most require tight coupling of processes with sharing that is not address-independent. None of the above systems support flexible client-controlled coherence, client-controlled memory placement (due to their address-dependent nature), or anytime updates.

Quasi-Caching [4] is very relevant to the topic of maintaining client-controlled coherence between a data source and cached copies of the data. Quasi-Caching assumes that each object has a computer that stores the most recent version and other computers store quasi-replicas that may diverge. They considered both time-based and scalar value based divergence (coherence) models. In their work, the allowed divergence is specified by the user in a fixed manner. The quasi-caching work describes when to update a client's cached copy but does not deal with the issue of how to do so efficiently. Furthermore, this work also does not support dynamically modifying the coherence model, or client-controlled memory placement.

Computer Supported Collaborative Work (CSCW) systems [48, 17, 14] share some of the features of *InterAct* (supporting interactive sessions across independent and potentially heterogeneous systems, update notification, etc.). However, most of these systems are tailored to a specific application, like cooperative engineering design, or distributed meetings. This has lead to a proliferation of isolated tools with little or no inter-operability.

There has also been some recent work on distributed data mining systems. The Kensington [21] architecture treats the entire distributed data as one logical entity and computes an overall model from this single logical entity. The architecture relies on standard protocols (such as Java Database Connectivity (JDBC)) to move the data. The Intelliminer [50] and Papyrus systems [20] are designed around data servers, compute servers, and clients as is the system presented in this work. All of the above systems rely on a message-passing-like interface for programming distributed data mining applications. *InterAct* provides a shared-object interface with features such as client-controlled coherence, memory placement, and anytime updates. In this paper, we limit ourselves to evaluating *InterAct* on client-server data mining applications although the *InterAct* system can be used for a broader class of distributed applications.

4 Runtime Framework

Shared data in **InterAct** are declared as *complex-object*s. Complex-objects are composed of *nodes* that may be linked together. Nodes may be C-style `struct`s, basic types, or some predefined *InterAct* types. Complex-objects could include recursive data structures such as graphs, trees, lists, arrays or collections of nodes. In Figure 1, we describe a general-purpose interactive mining algorithm mapped onto *InterAct*. In this example, the client has mapped three complex-objects, an array, a directed acyclic graph (DAG), and a list representing the shared data summaries, onto the virtual shared dataspace. An element in the list points to the DAG, represented by the connection. The server is responsible for creating and updating these data summaries. The client specifies a coherence model when mapping the summaries, synchronizes when required, and is responsible for the interactive querying component.

In *InterAct*, every complex-object moves through a series of consistent states, or versions. When a client first maps a shared complex-object, it specifies the desired coherence model. *InterAct* obtains a copy of the complex-object from the server on the first client access. At the beginning of each semantically meaningful sequence of operations, the client performs a synchronization operation, during which the system ensures that the local copy of the complex-object is "recent enough", as determined by the specified coherence protocol. If not, it obtains a new version from the server.

One of the principal innovations in the runtime system is the provision of the ability to allow each individual process to determine when a cached copy of a complex-object is "recent enough", through the specification of one of a set of highly relaxed coherence models. Changes to complex-objects must be made using mutually exclusive access. As long as applications adhere to these synchronization requirements, *InterAct* transparently handles all client-server communication including intra-object consistency and coherence maintenance.

InterAct defines the following coherence models. **One-Time Coherence** specifies a one-time request for data (complex-object) by the client. No history need be maintained. This is the default coherence type. **Polled Coherence** indicates that the client may request a current version when desired. The server may then attempt to reduce communication requirements by keeping track of the staleness of the data cached by the client. **Immediate Coherence** guarantees that the client will be notified whenever there are any changes to the mapped data. **Diff-based Coherence**

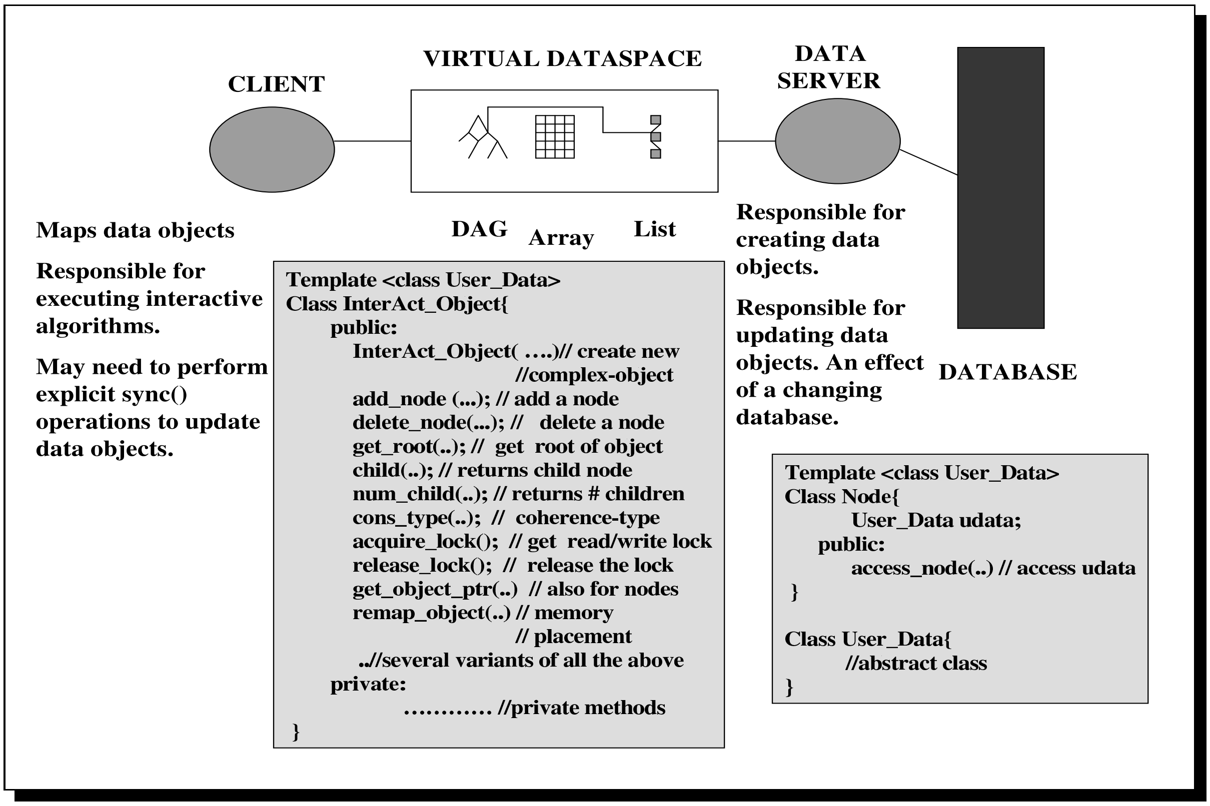

Figure 1: Interactive Client-Server Mining

guarantees that no more than $x\%$ of the nodes comprising a complex-object is out of date at the time of synchronization. **Delta Coherence** [46] guarantees that the complex-object is no more than x versions out of date at the time of synchronization. **Temporal Coherence** guarantees that the complex-object is no more than x real-time units out of date at the time of synchronization. In all cases, x can be specified by the client. Section 5 details the use of the different coherence models by various data mining techniques.

The coherence models are motivated by the ability of a large number of data mining and other interactive applications to tolerate a certain level of data staleness. In all cases, the interface involves the use of synchronization in order to bring data up-to-date, or to make modifications to the data. The application programming interface (API) allows processes to acquire a read or a write lock on shared data. A read lock guarantees that the shared data is up to date subject to the coherence model requested. A write lock always guarantees strict coherence.

In order to provide client-controlled memory placement, our API provides primitives by which each process (clients or servers) can locally remap nodes within a complex-object to improve spatial locality. *InterAct* transparently handles the remapping as a byproduct of supporting address independence. We next discuss the *InterAct* API.

4.1 Interface

In Figure 1, we describe the current interface available to the user within the two grayed rectangular regions. Our interface is essentially a set of template classes. `Node` is the template in which user data (using `Class User_Data`) can be embedded and from which it can be accessed (using `access_node`). `InterAct_Object` is the template with predefined functions for creating a complex-object, remapping (`remap_object`) it in memory in a locality-enhancing manner, adding (`add_node`) nodes to an *InterAct* complex-object, and deleting (`delete_node`) nodes from an *InterAct* complex-object. In addition, there are various methods that allow one to access the root node (`get_root`) of an *InterAct* object, identify (`num_child`) how many nodes are connected to a given node within a complex-object, and access such children nodes (`child`).

There are also functions for synchronizing (`acquire/release read/write lock` [1]) and modifying the required coherence type (`cons_type`). The User_Data class is used as a base to define what a node contains. The node may be composed of basic data types and pointers to other complex-objects. The interface requires the user to identify these special pointers (`object_ptr`, not in figure) to the system. We next describe how this interface can be used to create, map, and manipulate summary structures.

4.1.1 Example Usage

Server code: In this code fragment (Figure 2) the contents of each node within the complex-object to be created is first declared. This is done by deriving a sub-class

[1] Note that our implementation of a write lock is *relaxed*, in the sense that readers are permitted during a lock held in write mode.

```
Class lattice_node: public User_Data{
        int id; //item id
        int count; // support
}
Main(){
/****initialization routines ********/
/*create new InterAct object called lattice*/
InterAct_Object <lattice_node> *at =
        new InterAct_Object ("Lattice");
/*Obtain a write lock on the new object */
at→acquire_lock(write);

/*Insert  itemsets to Lattice */
Add_itemset(at,itemset,support);

/*Update  itemsets to Lattice */
Update_itemset(at,itemset,support)

/*Release write lock */
at→release_lock()
}
```

```
Add_Itemset(at, itemset){
/*get root node and add new child node */
Node <lattice_node> *root = at->get_root();
Node<lattice_node> *child = at->add_node(root);
lattice_node *idata = child->access_node();

/*enter values for new itemset */
idata->id = itemset.id;
idata->count = itemset.count;
}
```

```
Update_Itemset{at,itemset){
Node <lattice_node> *root = at->get_root();
for (I = 0; I < at->numchild(root); I++)
        if (at->child(root, I)→access_node()
                →id == itemset.id)
                break;
lattice_node *idata =
        at→child(root,I)→access_node();
/*Update support */
idata→count = itemset.count;
}
```

Figure 2: Server-Side Pseudo Code: *InterAct* method-calls shown in **bold**

```
Class lattice_node: public User_Data{
        int id; //item id
        int count; // support
}
Main(){
/****initialization routines ********/
/*map new InterAct object called lattice*/
InterAct_Object <lattice_node> *at =
    new InterAct_Object ("Lattice", sync, par);

/*Obtain a read lock on the new object */
at→acquire_lock(read);

/*Execute prioritized association query
  i.e. pick top 40 associations*/
Prioritized(at,at→get_root(),minsup,40);

/*Release write lock */
At→release_lock()
}
```

```
Prioritized(at, current, minsup, num_assoc)
/* if  required #assoc have been displayed, return*/
if(total_assoc >= num_assoc) return;
for (I = 0; I < at→numchild(current); I++){
  /* get  data for each child */
  lattice_node * child_data =
        at→child(root, I)→access_node();
 /* if child has support value greater than minsup*/
  if(child_data→count > minsup)
  /* if child has not yet been visited  enqueue on
        priority queue*/
    if (!visited(child_data)){
        PQ.enqueue(at→child(current,I)); }
} /*end for */

while(PQ not empty){
    Cnode = PQ.dequeue();
    print(Cnode); /*display association */
    total_assoc++;
    if (total_assoc >= num_assoc) return;
    /*recursively invoke Prioritized */
    Prioritized(at,Cnode, minsup,num_assoc)
    if (total_assoc >= num_assoc) return;
} /* end while */
}/*end Prioritized */
```

Figure 3: Client-Side Pseudo Code: *InterAct* method-calls shown in **bold**

called `lattice_node` from the abstract class `User_Data` as shown. Each `lattice_node` contains an itemset id and a count for that itemset. The main program in the server initializes the *InterAct* framework and formally declares and creates an *InterAct* complex-object called Lattice (using the above defined `lattice_node`). The server can then add, update, and delete (not shown) nodes after acquiring the necessary lock on the complex-object.

For expository simplicity both the the AddItemset and UpdateItemset pseudo-codes assume that the tree is only one-level deep, i.e. only children of the root node have to be added or updated. The AddItemset procedure first identifies the root of the complex-object using the `get_root` method, and then adds a child to the root using the `add_node` method. The data within the child node can then be accessed and added to by using the `access_node` method. The UpdateItemset procedure first identifies the appropriate child node of the root node that needs to be updated (using the `numchild` and `child` methods). It then accesses and updates the corresponding child node using the `access_node` method.

Client code: Like the server, the client also has to define the `lattice_node`. After performing the initialization operations the client can map a given object as shown in Figure 3. The call to create a new *InterAct* Object specifies the coherence type (`sync`) and coherence parameters (`par`) if any in addition to specifying the object name. This call examines a file/database with information associating each object name with a server (IP address and port number). If the process is determined to be a client, the necessary communication with the server is performed in order to retrieve the initial copy of the object. Once the object is mapped it can be synchronized with the server version by acquiring the appropriate lock. In the pseudo-code shown in Figure 3, the client invokes a procedure called Prioritized after acquiring a read_lock. After the procedure has been completed the client releases the read_lock.

`Prioritized` is a recursive procedure (described in [33, 1]) that identifies the X ($X = 40$ in the figure) most frequent associations and displays them. Each call to the procedure involves accessing the children of the current node and evaluating their support counts against the minimum support criteria (using the `child` and `access_node` methods). Nodes representing itemsets that meet the minimum support criteria, are added to the priority queue (priority is determined by support counts). Then the priority queue is dequeued as many times as the number of associations ($X = 40$) requested. This procedure uses the `child, numchild,` and `access_node` methods to access and read the data contained within the complex-object.

4.2 Memory Management and Access

Like any distributed object-based system, our interface identifies any pointers and their associated types to the runtime system in order to provide address independence. During complex-object creation, the complex-object's internal representation is divided into data and connection pages[2] (this division is transparent to the user as long as the defined templates are used to declare and access shared data).

Data pages contain all the nodes for a complex-object. Nodes are created and allocated as fixed-size structures so that array index arithmetic [3] may be used to access them efficiently (allowing variable sized nodes would simply involve a slightly less efficient access mechanism to the node). Separating connection information for a node (number of nodes linked to) allows the number of links to be variable and enables us to use array arithmetic on the data pages. Each node contains a single pointer into the connection pages that identifies the set of nodes that node links to. Information in the connection pages identifies the nodes within the complex-object in terms of an index, making address independence feasible. Separating connection and data information also co-locates all the pointers, enabling the runtime system to perform efficient pointer swizzling [54].

Laying the data out in semi-contiguous order facilitates efficient address to node

[2]Each complex-object is mapped to a disjoint set of pages, which enables our system to transparently detect changes to objects using virtual memory hooks. Since we are dealing with applications where the complex-objects are reasonably large relative to the size of a page, this does not result in memory wastage.

[3]Note that since complex-objects can dynamically change in size, all the pages for a complex-object need not be contiguous, so a slightly modified form of array indexing is needed.

mappings as long as node sizes are fixed, resulting in fast identification of changes to a complex-object as outlined below in Section 4.3. The use of node identifiers coupled with the above scheme also enables fast node to address mapping that permits us to update node changes rapidly as well as maintain mappings independent of server mappings, as outlined below in Section 4.7.

4.3 Object Modification Detection

The technique we use to detect modifications is similar to that used by multiple-writer page-based software distributed shared memory systems [5, 8], except that we use a node as the granularity at which we detect modifications. At the start of every acquire of a write lock (see Figure 4), all relevant complex-object pages are marked read-only using the *mprotect* virtual memory system call. When a processor incurs a write fault, it creates a write notice (*WN*) for the faulting page and appends the *WN* to a list of *WN*s associated with the current interval, or region encapsulated by an acquire and a release. It simultaneously saves a pristine copy of each page, called a *twin*, and enables write permissions [8]. When the lock is released, the *twin* is used to identify the nodes modified within the interval.

At the release, all objects that have been modified (identified through the *WN* list) increment their associated object timestamp (or *version* number). These objects are efficiently identified since our *WN* list is maintained as a hash table containing the <page address, object identifier> pairs. Since we ensure that complex-objects reside on separate pages, a write to a page corresponds to a write to a single complex-object. Modified nodes are identified by comparing the modified pages to their *twins*. Comparison is thus limited only to those pages that are actually modified. These modified nodes are then communicated to the object manager along with the latest version number in order to keep the manager's copy up-to-date. The object manager has a timestamp (or version) map associated with each object. A timestamp map contains an entry for each node indicating the last time it was modified (see Figure 4). Upon receiving modifications, the manager updates its copy of the data as well as of the timestamp map.

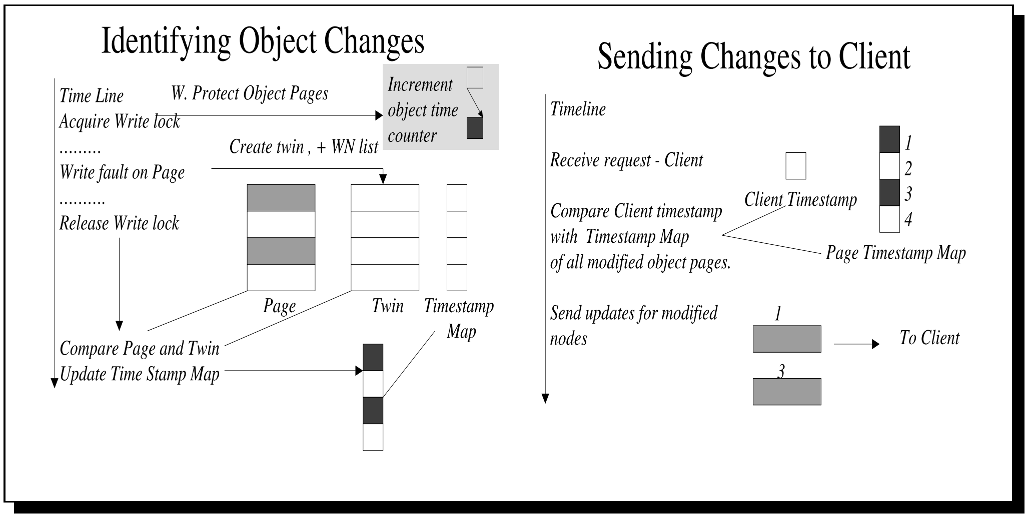

Figure 4: Efficient Shared Data Updates

4.4 Updating an Object

When asked for changes by the caching process, the object manager (server) compares its timestamp map for the complex-object against the last time the client has been appraised of an update. The result of the timestamp comparison is a run-length encoding of the node data and node connections that have been modified, which constitute the *diff* of the complex-object (Figure 4). Header information also specifies the complex-object identifier, and number of node data and connection updates.

On the client side, we maintain information corresponding to the objects that are mapped and where they are stored in local memory. On receiving a diff message, we update the corresponding object by decoding the header to determine the object identifier. This object identifier is then used to determine the local location of the object. Data and connection information for nodes within the object are similarly address independent.

The issue of containment (or aggregation) in an object-oriented design is an important one for performance and correctness reasons. In our current implementation, when a client maps a particular object, we update all nodes that belong to the particular object immediately. However, if a node within this object points to another object or another node within another object, that object is not copied immediately. It is copied lazily upon client access or client request. The *InterAct* interface implicitly gives the programmer control over what needs to be copied immediately and what can be lazily copied. All the programmer has to do is to create separate complex-objects in this case.

4.5 Generating Anytime Updates

If a client request comes in while a write lock on the object is held by the server or managing process, one approach would be to wait[4] till the server transaction commits before sending an update to the client. This may not be acceptable for applications requiring a quick response, especially when the lock is held for a long time. Our approach is to twin the entire object on creation. When the write lock is released, the object twin is updated using the *diff* mechanism described above. If a client request comes in during a write lock, the system returns the update from the twin rather than the original object. During the application of the *diff* on the twin, the system returns the update from the original object. This ensures that the client rarely has to wait for the requested data in practice. This approach is costly in terms of space. An alternative approach which we considered is to *twin* only the pages modified (as described in Section 4.3) during the write lock and deliver updates for modified pages from these twins. However, this approach involves pointer tracking and associated synchronization and therefore is more costly in terms of time. One could potentially switch between the two approaches as a function of application requirements and system configuration.

4.6 Coherence Maintenance

The runtime system optimizes data communication by using the coherence model specified by the user to determine when communication is required. The goal is to reduce messaging overhead and allow the overlap of computation and communication. Implementation of the **Immediate Coherence**, **Polled Coherence**, and **One-Time Coherence** guarantees are fairly straightforward. **One-Time Coherence** does not require any meta-data (timestamp) maintenance, nor does it require

[4] In order to guarantee an atomic update, the data cannot be sent in an as is condition, as partial changes to the object may have occurred concurrently.

the server to generate object diffs. **Polled Coherence** is implemented by having the client send the most recent timestamp of the object it has seen. Under **Immediate Coherence** the server keeps track of this information. Under both Polled and Immediate coherence only those nodes with timestamps greater than this value are communicated. The difference between Polled and Immediate coherence is that in the latter the server notifies the client when a complex-object has been modified while the former has no such notification protocol. This allows the runtime system under Immediate coherence to check for notification messages before issuing an update request to the manager (server) eliminating some communication traffic. The upside to Polled Coherence is that the server does not need to maintain client-specific state on a per complex-object basis.

Temporal Coherence is supported by having the runtime system on the client's side poll for updates every x time units, as defined by the user. To keep track of **Diff-based Coherence**, the server maintains a cumulative count of nodes per complex-object that are modified since the last client update. If this cumulative count exceeds a preset user-defined value (referred to as the *diff* parameter), the client is sent a notification (similar to Immediate Coherence) and a subsequent update. **Delta Coherence** is kept track of in a manner similar to **Diff-based Coherence**. In these cases as well, the server has to maintain the last timestamp seen by the respective clients.

4.7 Memory Placement

Different clients may have different mining agendas, leading to different data structure access. *InterAct* permits the clients to place the mapped data structure in memory in a locality enhancing manner by using the remap()[5] function. For example, if the structure is a tree and most of the client interactions are going to induce a breadth-first evaluation of the tree, then the tree can be placed in memory in a breadth-first fashion to improve cache locality. *InterAct* currently supports breadth-first, depth-

[5]Clients need to execute this only once. Subsequent updates from the server are automatically handled correctly by our system's address translation mechanisms.

first, and user-defined placement [38]. User-defined placement allows the programmer to define a condition that splits the nodes in an object into separate sets of contiguous memory.

5 Applications

In past work [37, 39, 40, 33], we have shown that it is possible to design useful summary structures for several mining applications so that subsequent queries can operate on these summary structures rather than the actual data. Within our framework of remote mining, these summary structures are generated and kept up-to-date by the data server, and subsequently mapped and operated on by the client. In this work, we simulate the updates on the server side according to real data and application update properties, described below.

Association and Sequence Mining: Given a database of transactions where each transaction consists of a set of items, association discovery[2] finds all the item sets that frequently occur together, and also the rules among them. Sequence discovery essentially involves association discovery over temporal databases. It aims to discover sets of events that commonly occur over a period of time.

For association and sequence mining, the summary structure we use is the Itemset Lattice [1, 33] (or Sequence Lattice [39]), which contains pre-mined patterns and the corresponding support[6] information. Responses to user requests typically involve computing a constraint-based subset of the entries in the lattice. Updates to the lattice are handled as described in [52]([39]). Each incremental update, reflecting new data, typically combines multiple actual transactions, for performance reasons [52, 33]. There are two possible kinds of updates to the summary structure based on the type of mining being performed. When mining is performed on the entire database, new transactions are usually only added to the database — we call these additive updates. When mining is performed on a window of transactions in the database, changes in the database result in almost as many additions as deletions to the window

[6]Support is the number of times the pattern occurs in the dataset.

— we call these windowed updates. Both types of updates typically result in changes to anywhere from 0.1% to 10% of the summary data structure. This is because in typical scenarios the number of transactions being added or deleted is a small percentage (0.1%-1%) of the total number of transactions being represented (and in the case of sequence mining not all customers are part of each update), so the net impact on the summary is relatively low. Additive updates mostly result in modifications to support counts and a few pattern additions and deletions. However, for windowed updates, the changes tend to result in more associations being added and/or deleted. For these applications, since the user is usually interested in keeping track of less frequently occurring associations or sequences, a stricter coherence model such as **Polled Coherence** or **Immediate Coherence** is generally preferred.

Discretization: Discretization has typically been thought of as the partitioning of the space induced by one (say X) or more continuous attributes (base) into regions (e.g., $X < 5$, $X >= 5$), which highlights the behavior of a related discrete attribute (goal). Discretization has been used for classification in the decision tree context and also for summarization in situations where one needs to transform a continuous attribute space into a discrete one with minimum "loss". Our program is an instance of 2-dimensional discretization (two base attributes, described in [37]).

Interactions supported include generating an optimal discretization (based on entropy or classification error), and modifying the location and number of control points (which partition the two-dimensional base attribute space). The summary structure required to support such interactions efficiently is the joint probability density function (pdf) of the base and goal attributes. This pdf is estimated at discrete locations. While several techniques exist to estimate the density of an unknown pdf, the most popular ones are histogram, moving window, and kernel estimates [13]. We use the histogram estimate described in [13]. The advantage of this estimate is that it can be incrementally maintained in a trivial manner (a histogram estimate is essentially the frequency distribution normalized to one). Moreover, the more complicated kernel estimates can easily be derived from this basic estimate [13]. Each update corresponds to one transaction and every update modifies exactly one entry in the array. Each update is simply a small perturbation on the pdf estimate and as such does not affect the quality of discretization significantly. Thus, this technique would benefit from using **Diff-based Coherence** without affecting the quality of the results. The *diff* parameter specifies the amount of data that needs to change before it is significant to the application.

Similarity Discovery in Datasets: This application computes and maintains the similarity between two or more datasets. Such measures of similarity are useful for clustering homogeneous datasets. In [40], we define the similarity between two datasets to be a function of the difference between the set of associations induced by them, weighted by the supports of each association. To compute and maintain the similarity between n datasets, the client maps the itemset lattices (containing the association patterns and their supports) from each of the distinct data sources and then computes the pairwise similarity measures.

It has been noted [12] that incorporating domain bias in the similarity measure via suitable interactions can be very useful. In this application, we support the following operations: similarity matrix re-generation after a data structure update, identifying influential attributes, and constraining the similarity probe set via constraint queries on the itemset lattices. Incrementally maintaining the association lattices has already been discussed above. However, since this application is more interested in general patterns, even if a large percentage of the mapped summary structure is modified over a period of several updates, it has been shown that the percentage change in the measured similarity is not significantly affected. The measured similarity directly correlates more to the magnitude of the change in data. This magnitude is not directly measurable without a large amount of overhead. However, the use of **Delta Coherence** captures this application's requirements by allowing the data to be several versions out of date without affecting result quality.

Other Applications: We have described how one can define summary structures for four key data mining tasks. The same principles may be applicable to other data mining tasks as well and this issue is under investigation. For instance, for clustering one may choose to create, maintain (server-side), and map (client-side) cluster descriptors. For decision-tree classifiers one could either create, maintain, and map the actual trees themselves or a compact representation of the tree such as an FFT representation [27]. For both neural networks and bayesian networks the actual network descriptors (weights and links) could be mapped.

Application	Object Size	UPS	%Change	Change Type
Association Mining	3.3MB	0.5	4%	ADD/DEL
Sequence Mining	1.0MB	1	10%	MOD/ADD
Discretization	0.5MB	100	0.002%	MOD
Similarity(1)	0.5MB	5	10%	MOD/ADD
Similarity(3)	3X0.5MB	5	0.33%	MOD/ADD

Table 1: Server Update Properties

Properties of Server Updates We outline the exact nature of the updates[7] used for our experiments for each application in Table 1. The second column refers to the size of the summary structure when the server starts up. UPS corresponds to updates per second on the server side. The column labeled %Change corresponds to the percentage of nodes in the summary structure that are changed over a single update, and are representative of realistic workloads for each of the applications. For Similarity Discovery, we computed the similarity among four databases. Updates on one of these databases (Similarity(1)) had different properties from the other three (Similarity(3)). The last column in the table refers to the dominant change type of the given update. ADD refers to the fact that the update adds new nodes, DEL refers to the fact that the update deletes nodes, MOD refers to the fact that the update modifies existing nodes. The order in which change types appear in column five of the table are in decreasing order of dominance. For example, ADD/DEL refers to the fact that on average, executing the corresponding update results in more additions than deletions to the summary structure.

[7] The datasets used are described in [41].

6 Experimental Evaluation

We evaluate our framework in a distributed environment consisting of SUN workstations connected by 10 or 100 Mbps switched Ethernet. Unless otherwise stated, the clients use a 100 Mbps link, and are 270 MHz UltraSparc IIi machines. The clients in each application interact with the server by sharing the summary data structures with the server. The server creates the summary data structure and updates it corresponding to changes in the database (which we emulate). The client maps these data structures using one of the provided set of coherence models. Updates are then transmitted to the client according to the coherence model chosen.

6.1 Runtime Overhead

We evaluated the overhead imposed by our system on the server side during normal execution without client connections. To evaluate this overhead, we compared the test application written using the *InterAct* interface against a program written using standard C++. Using UNIX malloc with the C++ program, we found that the application running on top of our framework executed slightly faster due to our improved memory allocation policy. We use a special memory placement library [38] in our system that improves locality and therefore cache performance. Comparing against the standard C++ program with calls to the UNIX malloc replaced by calls to our memory placement library, we found the runtime overhead imposed by using our template interface (which involves more indirect accesses) to be less than 5% for our application suite.

6.2 Client-Side Caching

In typical client-server applications, the client makes a request to the server, the server computes the result, and then sends the result back to the client. Since the interactions in our applications are often iterative in nature, caching the data structure on the client side so that repeated accesses may be performed locally eliminates overhead and delays due to network latency and server load. The potential gain from client-side caching depends on a number of factors: the size of the shared data, the speed of the client, the network latency, and the server load.

In this experiment, we ran each of our applications under the following scenarios:

1. Client-Side Caching (CSC): the client caches the summary structure and executes the query on the local copy.

2. Server Ships Results to Client (SSRC): the client queries the server and the server ships the results back to the client. This scenario is similar to the use of an RPC mechanism. In order to better understand the impact of server load, we varied the number of clients serviced by the server from one (SSRC) to eight (**L**oaded-SSRC). For each of the applications considered, Associations, Sequences, Discretization, and Similarity, the average size of the results shipped by the server was 1.5MB, 0.25MB, 0.4MB and 0.75MB respectively.

We measured the time to execute each query under both scenarios. We evaluated each scenario on client machines that were either an UltraSparc (143Mhz) machine or an UltraSparc IIi (270Mhz). In each case, our server was an 8-processor 333 MHz UltraSparc IIi machine. Results are presented in Tables 2 and 3 for these scenarios under different network configurations. We varied the network configuration by choosing clients that are connected to the server via a 10 Mbps or a 100 Mbps Ethernet network.

Application	Client(143)			Client(270)		
	CSC	SSRC	L-SSRC	CSC	SSRC	L-SSRC
Association	2.4	1.6	2.5	1.5	1.4	2.3
Sequence	0.58	0.55	0.86	0.35	0.5	0.73
Discretization	0.87	0.67	1.08	0.55	0.6	0.98
Similarity	0.35	0.55	0.98	0.11	0.37	0.94

Table 2: Time (in seconds) to Execute Query: 100Mbps Interconnect

Application	Client(143)			Client(270)		
	CSC	SSRC	L-SSRC	CSC	SSRC	L-SSRC
Association	2.4	4.05	7.2	1.5	2.5	5.1
Sequence	0.58	0.85	1.35	0.35	0.63	1.18
Discretization	0.87	1.35	2.75	0.55	0.94	1.6
Similarity	0.35	1.5	2.7	0.11	0.9	2.4

Table 3: Time (in seconds) to Execute Query: 10Mbps Interconnect

The results in Table 2 and Table 3 show that client-side caching is beneficial for all but a few of the cases. In particular, the following trends are observed. Client-side caching is more beneficial under the following scenarios: the network bandwidth is low (speedups from client-side caching under the 10Mbps configuration are larger (1.5 to 23) than the 100Mbps numbers (0.6 to 9)), the server is loaded (comparing the L-SSRC column (speedups of 1.1 to 9) with the SSRC column(speedups of 0.6 to 3.5) with a 100 Mbps network), the client is a fast machine (comparing the columns involving the 270Mhz client versus the slower clients), or the time to execute the query is low (comparing the row involving similarity discovery with the row involving association mining). In other words, as expected, the benefits from client-side caching are a function of the computation/communication ratio. The lower the ratio, the greater the gain from client-side caching.

These results are presented just to illustrate the fact that *InterAct* enabling such caching is very useful for such applications especially when deployed on the Internet. The results from this experiment underscore two key aspects. First, *it is possible* to design useful summary structures that summarize the dataset effectively for mining purposes. Such summary structures can be accessed efficiently to answer a set of useful queries rapidly and efficiently and are significantly smaller than the original datasets they summarize. Second, shipping the summary structure to the client offloads much of the computational work from the server and accelerates the query processing by eliminating the client-server network delay.

6.3 Coherence Model Evaluation

In this section, we evaluate the benefits of using relaxed coherence models. In our experiments, the clients map the shared summary structure, perform iterative requests simulating a realistic data mining interaction, and synchronize with the server periodically. The server concurrently updates the shared data structure, reflecting changes to the actual data. Figure 5 reports the average synchronization time (time for a read `acquire_lock`, defined in Section 4.1, and used to bring the complex-object up-to-date according to the desired coherence model) for each of the applications under different coherence models. In these experiments, all clients use the same coherence model. We measured the synchronization time over a window of several (35 to be precise [8]) synchronizations and averaged this time. This average synchronization time in seconds is represented on the Y axis. The X axis corresponds to the number of clients in the experiment.

Tables 4 through 7 present a breakdown of the total number of requests made by clients, and the total amount of data communicated under each of these coherence models for each of the applications. The first column represents the number of clients in the system. The subsequent columns represent the cumulative sum of the data sent out by the server to all the clients and the total number of requests made by all the clients, with each client performing 35 synchronization operations, for each of the coherence models evaluated for the application.

The average synchronization time can be broken down into two components, the communication overhead and the time spent waiting at the server. Due to limited resources, we could evaluate our work only on up to eight clients. Since the server is multi-threaded and has up to eight processors, we see very little increase in server load overheads with an increase in the number of clients. However, as the number of clients serviced by the server increases, the average synchronization times increase due to contention for network resources, as well as due to the fact that the server modification window is also consequently higher. This underscores the need to reduce

[8]Going to larger window sizes did not affect the average synchronization times for our workloads.

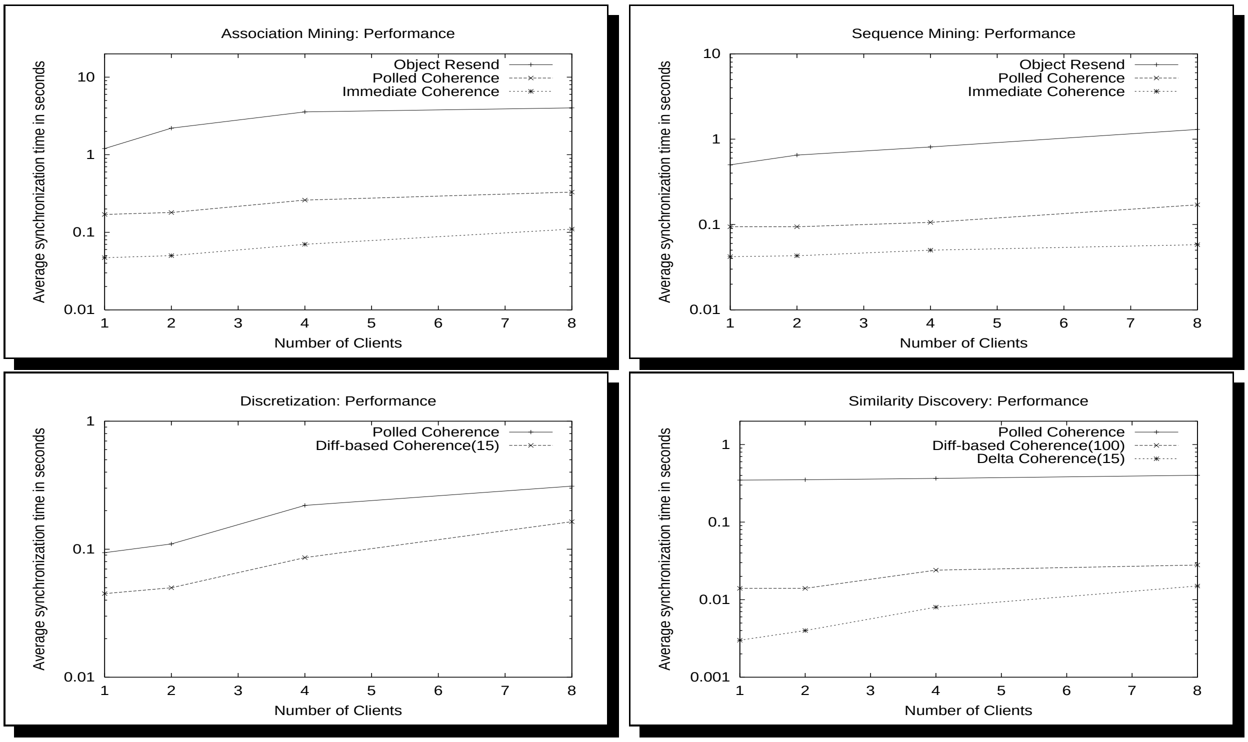

Figure 5: Coherence Model Evaluation

communication and server load.

In order to evaluate the effect of the coherence model on performance, we begin by evaluating the effectiveness of using diffs (using **Polled Coherence**) as opposed to resending the entire complex-object (using **One-Time Coherence**), the strawman approach taken by most existing commercial object-oriented systems [9]. We found that sending diffs could be 10-30 times faster than resending the entire complex-object for reasonable-sized (10% of nodes) changes made by the server. The size of the messages used to update the clients is much smaller, resulting in reduced communication overhead. The gains due to reduction in communication cost also reflect the reduction in overhead from client-server flow control due to finite buffer sizes.

We next evaluate the impact of having the flexibility of using different client-

[9] Object techniques involving re-executing methods on all cached copies are not possible on data mining applications, since updates use large amounts of I/O, with the data residing only on the server. Hence, these approaches are not suitable for such applications.

#C	Polled		Immediate	
	Total Data	# Requests	Total Data	# Requests
8	145306×32	280	141989×32	70
4	74552×32	140	74552×32	35
2	37276×32	70	37276 ×32	16
1	18638×32	35	18638×32	7

Table 4: Association Performance Breakdown

#C	Polled		Immediate	
	Total Data	# Requests	Total Data	# Requests
8	85414×40	280	33172×40	140
4	43402×40	140	18299×40	53
2	20410×40	70	9044×40	34
1	9744×40	35	4668×40	14

Table 5: Sequence Mining Performance Breakdown

#C	Polled		Diff(15)	
	Total Data	# Requests	Total Data	# Requests
8	2013×24	280	1640×24	77
4	973×24	140	896×24	44
2	393×24	70	374×24	18
1	207×24	35	178×24	8

Table 6: Discretization Performance Breakdown

#C	Polled		Diff(100)		Delta(15)	
	Total Data	#Req	Total Data	#Req	Total Data	#Req
8	111415×32	1120	33792×32	69	18448×32	34
4	52213×32	560	15841×32	31	8920×32	17
2	25598×32	280	9976×32	17	4224×32	8
1	12840×32	140	4224×32	9	2112×32	4

Table 7: Similarity Performance Breakdown

controlled coherence models. For both association mining and sequence mining, we evaluated polled coherence and immediate coherence. Immediate coherence performs several factors better than polled coherence for these two applications. Polled coherence results in the client sending a request to the server at every synchronization point. Immediate coherence dictates that the server send a notification message to the client. On seeing the notification message at a synchronization point, the client polls the server for the update.

For association mining, the data communicated in both the immediate and polled cases is the same (Table 4). The synchronization rate is a function of the query processing time and the synchronization time. In this application, the query processing time dominates, resulting in the server modification window (the time between requests for updates) being roughly the same under both protocols. Additionally, the server updates in this application primarily involve adding and deleting nodes (see Table 1), explaining the total size of the changes being the same. However, there is a 75% decrease in the number of messages sent out under the former model. The synchronization rate is higher than the rate at which the server modifies data, resulting in unnecessary requests when using polled coherence. This results in a 3-fold performance improvement.

Sequence mining also sees a 3-fold improvement in synchronization time when using immediate coherence as opposed to polled coherence. There is a reduction not only in the number of requests but also in the total data communicated. In this application, the query processing time is small. Hence, when synchronization time is reduced, the server modification window is also smaller. In addition, this application primarily modifies existing nodes (see Table 1). These factors combine to reduce the amount of data. The synchronization rate in this application is fairly close to the rate at which the server modifies data. Hence, due to timing variations in receiving notifications with immediate coherence, some of the synchronization operations remain local and do not request updates, resulting in a 2-fold reduction in the number of requests.

6.3.1 Diff-based and Delta Coherence Model Evaluation

While association mining and sequence mining benefit from sending updates rather than the entire complex-object, they both require the client to have the latest copy of the shared data. However, discretization and similarity discovery can tolerate some staleness in the interaction structure without losing much accuracy.

Discretization can make use of the diff-based coherence model since we know that the quality of discretization is not affected by small changes in the shared data (we quantify this in the next paragraph). Similarity discovery does not benefit as much from diff-based coherence since every server update is likely to modify roughly 10% of one of the summary structures, or 1600 nodes (see Table 1), which far exceeds the *diff* parameter that we use. The number in brackets for diff-based coherence in Figures 5 and 7 is the *diff* parameter, or the number of nodes that must be modified to trigger an update to the client (set to 15).

However, similarity discovery benefits more from the delta coherence model due to the fact that the result quality is affected more by the magnitude of the change in data than by the amount of data that has changed. The magnitude of change per server update is small, while the number of nodes changed is about 10%. The number in brackets for delta coherence is the maximum number of server updates (or versions: corresponding to releasing a write lock) between updates to the client. For these applications, we compare the average synchronization times with polled

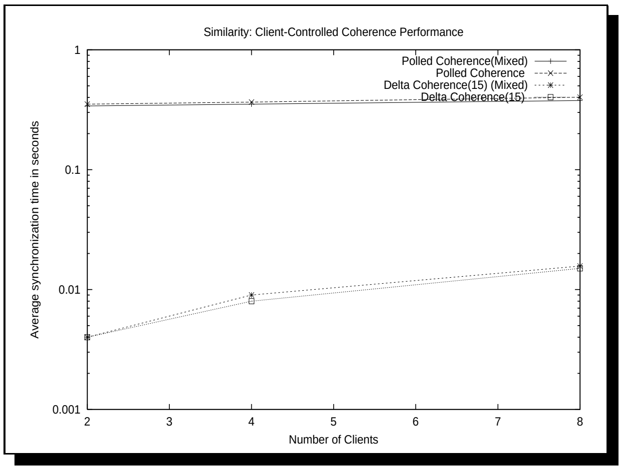

Figure 6: Effect of Client Controlled Coherence on Similarity Performance

coherence (object resends are always going to be worse than polled coherence).

Referring to Figure 5 for discretization, we find that diff-based coherence, when using a diff parameter of 15, was on average twice as efficient as polled coherence. This difference increases as the diff parameter is increased. For a diff parameter of 100 we found that the average synchronization time is 15-20 fold better. Delta coherence is particularly effective for similarity discovery, outperforming polled coherence by three orders of magnitude at low server load.

In both of these applications (referring to Tables 6 and 7) there is a reduction not only in the number of requests but also in the total data communicated. While part of the reason for this behavior is the same as discussed earlier for association and sequence mining, the main cause for this reduction is the applications' tolerance for staleness as specified by the relaxed coherence models.

6.4 Client-Controlled Coherence

An important contribution of our system is the fact that different clients may map the same shared structure using different coherence models. As an example, a client interested in similarity discovery involving a particular database could map the association lattice (as described in Section 5) of the database using the delta coherence model. The same lattice may be mapped by another client for the purpose of association mining using the polled coherence model.

In this experiment, we considered the following configurations. In the polled configuration, all eight of our clients used polled coherence. In the delta configuration, all eight used delta coherence. In the mixed configuration, four used delta coherence and four used polled coherence.

We found the average synchronization time of clients under polled coherence in the *Mixed* configuration to be slightly lower due to reduced traffic, and that for clients using delta coherence to be slightly higher due to the extra traffic from the clients using polled coherence. The average synchronization times of clients under delta coherence in the *Mixed* experiment were two orders of magnitude lower than those for when all the clients mapped the data using polled coherence. If client-controlled coherence were not used, the server would have to adhere to the strictest coherence model for correctness, in this case, polled coherence, resulting in much reduced performance. Thus, by using the coherence model required by each client, the server is able to improve overall performance.

6.5 Interaction Quality

The above results indicate that encoding knowledge about application behavior by choosing the appropriate coherence model can improve performance. This improved performance comes about due to reduced resource consumption (server processing time, low network utilization). However, to better understand the implications of lower resource usage one must also evaluate the corresponding loss in result quality for the application when using more relaxed coherence models. Such resource-aware computing issues have been studied in the context of other domains such as mobile-computing[43] and multimedia-computing[31, 32] and we apply a similar analysis below for discretization and similarity discovery.

For each of these two applications, we plotted the result quality under the different coherence protocols over a certain period of time (demarcated by server updates). Figure 7 presents these results. For discretization, the result quality is represented by

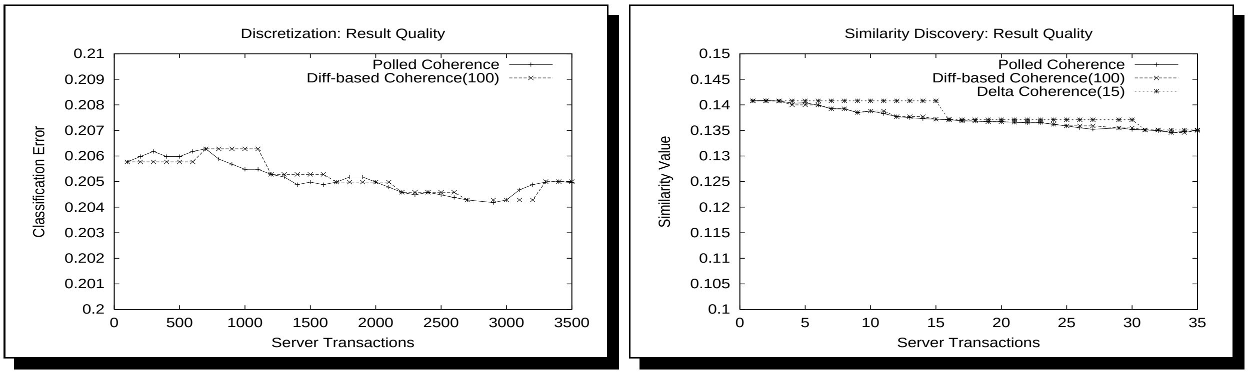

Figure 7: Result Quality

the classification error — the fraction of points in the space that are mis-classified. The plot for polled coherence represents the best achievable classification error for the algorithm. The plot for diff-based coherence (with a diff parameter of 100 as opposed to 15) is the error obtained when using a relaxed coherence model. Clearly, for this application the loss in quality is not significant. In fact, it is off by less than 1% of the exact error at all instances in time even for such a high diff parameter. For similarity discovery, the result quality is represented by the similarity value, which represents the distance between two datasets. The similarity value using delta coherence is no worse than 2% off the value obtained using polled coherence at all instances in time. For more details on the network-aware adaptive nature of these two application, readers are refered to a recent paper[34].

6.6 Effect of Modification Rate

In order to evaluate the impact of the server modification rate on our choice of coherence model, we modified the similarity discovery experiment described in the previous sections in the following ways. Two of the four lattices that we map have server update characteristics as described in Table 1, Similarity (1). For the other two lattices, we used the server update characteristics described under Similarity(3), where each server update modifies less that 0.1% of the data structure. Each lattice is maintained by a separate server process running on our 8-processor server.

We then evaluated the average synchronization time for one client in the system

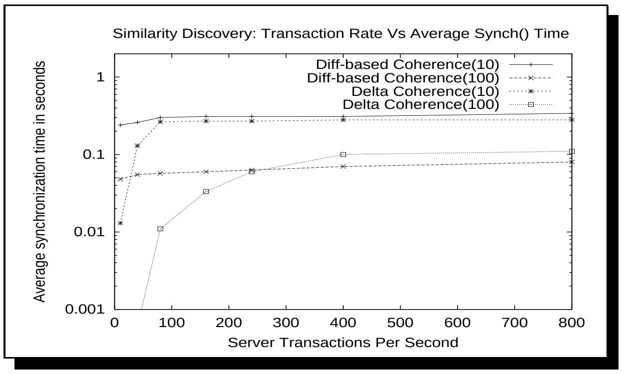

Figure 8: Effect of Transaction Rate on Similarity Performance

while varying the server updates per second. We varied *diff*, the diff parameter or the number of nodes that differ before an update is sent to the client, from 10 to 100 for diff-based coherence (see Figure 8). We also varied *delta*, the number of virtual time intervals between successive updates to the client, from 10 to 100 for delta coherence. The larger the diff/delta values, the lower the average synchronization time. The delta-based approach still does better for this application at a low server update rate since it minimizes the communication with the client. However, at larger update rates (crossover point 230 ups), diff-based coherence (diff_num = 100) begins to perform better than delta coherence. The reason for this is that at higher update rates, updates to the client are sent too frequently (in other words, a delta_num of 100 is too low) for all the complex-objects. However, since only two of the four lattices have a large percentage modified per update, for a *diff* value of 100, only these two lattices will cause updates to be sent to the client. The other two lattices are not modified at the same rate, resulting in lower average synchronization times when using diff-based coherence with a *diff* value of 100.

This experiment highlights the influence of server update properties on the choice of a coherence model, as well as the importance of being able to dynamically change coherence requirements as the application behavior changes. Choosing among relaxed coherence models is not only a function of what the application can tolerate in terms of data staleness, but also a function of how much and how often changes are made to the shared summary structure, in order to be effective.

6.7 Locality Enhancements

Locality enhancing memory placement is especially useful when the client uses a predefined traversal of the shared summary structure. We illustrate its benefits using association mining, where we found that for different queries, a different mapping of the data structure presented the best results. For example, when computing the most frequent associations (sometimes referred to as quantified associations) a breadth first representation of the lattice is most often desired since the more frequent an association is the closer it is likely to be to the root of the lattice. One can be more exact using the user-defined memory placement for this query. When computing inclusive associations whereby one desires to find all associations involving a particular item then the traversal is typically a depth-first traversal. In such cases one may want to use a depth first memory placement of the lattice. These different association queries are commonly used in online association mining [1, 33]. For these queries, we found up to a 20% improvement in execution times by remapping the data structure according to the best mapping strategy.

7 Conclusions and Future Directions

We have described a general runtime framework that supports efficient data structure sharing among distributed and interactive components of client-server applications. While the system is general enough to support a wide range of application domains, in this paper we have demonstrated the utility of the system for, and evaluated its performance on, a suite of interactive data mining 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. In the event that this shared data is modified, the complexity of determining exactly what data to communicate among clients and servers, as well as when that data must be communicated, is encapsulated within the runtime system. Each process has the ability to use the dynamically modifiable relaxed coherence mechanisms to encode application-specific knowledge about sharing behavior and requirements on a per data structure basis.

This information can then be used by the runtime to further reduce communication time potentially by several orders of magnitude. For those applications in our test suite that can take advantage of the more relaxed coherence protocols, we have also shown that the degradation in result quality is less than 2%. In addition, the any-time update and client-controlled memory placement features made possible by the address-independent virtual memory-based design of the runtime system help improve the performance of several data mining applications.

Further refinement of the system involves improved support for heterogeneous platforms as well as integration with tightly-coupled software shared memory systems [10, 42]. The latter integration will provide the ability for remote satellites to interact with computationally intensive components that require tightly-coupled parallel processing. The resulting system, called InterWeave [9], is on-going work and represents a merger and extension of our previous Cashmere [49] and *InterAct* systems, combining hardware coherence within small multiprocessors, Cashmere-style lazy release consistency within tightly coupled clusters, and *InterAct*-style version-based coherence for distributed shared segments.

Acknowledgements: Thanks to the anonymous reviewers for their useful feedback, which enabled us to improve the quality of this article.

References

[1] C. Aggarwal and P. Yu. Online generation of association rules. In *IEEE International Conference on Data Engineering*, February 1998.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining.* MIT Press, 1996.

[3] S. Ahuja, N. Carreiro, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[4] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM TODS*, 15(3):359-384, Sept. 1990.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[6] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, June 1992.

[7] M. Carey, D. DeWitt, J. Naughton, M. Solomon, et al. Shoring Up Persistent Applications.. *Proc. of the 1994 ACM SIGMOD Conference*, 1994.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] D. Chen, C. Tang, X. Chen, S. Dwarkadas and M. Scott. Beyond S-DSM: Shared State for Distributed Systems. URCS Technical Report 744, Univ. of Rochester, Mar 2001.

[10] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A Middleware System for Distributed Shared State. In Fifth Workshop on Languages, Compilers, and Runtime Systems (LCR) 2000, Rochester, NY, May 2000.

[11] D.E. Culler and J.P. Singh. Parallel Computer Architecture – A hardware/software approach. Morgan Kaufmann, 1999.

[12] G. Das, H. Mannila, and P. Ronkainen. Similarity of attributes by external probes. In *Proceedings of the 4th Symposium on Knowledge Discovery and Data-Mining*, 1998.

[13] L. Devroye. A course in density estimation. In *Birkhauser:Boston MA*, 1987.

[14] P. Dewan and J. Riedl. Towards computer-supported concurrent software engineering. In *IEEE Computer, Volume 26, Number 1*, 1993.

[15] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of 5th WWW Conference (journal version: IJCN)*, 1997.

[16] D. Grunwald, B. Zorn and R. Henderson. Improving the cache locality of memory allocation. In *Programming Languages Design and Implementation*, June 1993.

[17] G. Fitzpatrick, S. Kaplan, and W. Tollone. Work, locales and distributed social worlds. In *European Conference on Computer Supported Collaborative Work*, 1995.

[18] M. J. Franklin. Client Data Caching: A Foundation for High Performance Object Database Systems. Kluwer Academic Publishers, 1996

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[20] R. Grossman, S. Bailey, S. Kasif, D. Mon, A. Ramu, and B. Malhi. Design of Papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[21] Y. Guo, S. Rueger, J. Sutiwaraphun, and J. Forbes-Millot. Meta-learning for parallel data mining. In *Proceedings of the Seventh Parallel Computing Workshop*, 1997.

[22] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *High Performance Computer Architecture*, pages 14–25, February 1996.

[23] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client-Server Architectures. IEEE Conf. on Parallel and Distributed Information Systems, 1994.

[24] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[25] A.D. Joseph, A.F. deLespinasse, J.A. Tauber, D.K. Gifford, and M.F. Kaashoek. Rover: A toolkit for mobile information access. In *15th SOSP*, Dec 1995.

[26] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[27] H. Kargupta, B. Park, D. Hershberger, and E. Johnson. Collective data mining: A new perspective toward distributed data analysis. In *Advances in Distributed and Parallel Knowledge Discovery, Kargupta and Chan ed.*, 2000.

[28] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, pages 401–407, November 1994.

[29] L. Kontothanasis, G. Hunt, R.Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *PROC of the 24TH ISCA*, June 1997.

[30] B. Liskov et al. Safe and Efficient Sharing of Persistent Objects in Thor. In *SIGMOD*, 1996.

[31] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1994.

[32] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware Resource Manangement for Distributed Multimedia Applications. In *Journal on High-Speed Networking*, 1998.

[33] S. Parthasarathy. Active Data Mining in a Distributed Setting. PhD Dissertation, University of Rochester, 1999.

[34] S. Parthasarathy. Towards Network-Aware Data Mining. In *International Workshop on Parallel and Distributed Data Mining*, along with IPDPS 2001.

[35] S. Parthasarathy and S. Dwarkadas. InterAct: Virtual sharing for interactive client-server applications. In Fourth Workshop on Languages, Compilers, and Runtime Systems (LCR), May 1998.

[36] S. Parthasarathy and S. Dwarkadas. Shared State for Client Server Mining. In SIAM International Conference on Data Mining, 2001.

[37] S. Parthasarathy, R. Subramonian, and R. Venkata. Generalized discretization for summarization and classification. In *PADD*, January 1998.

[38] S. Parthasarathy, M. Zaki, and W. Li. Memory placement techniques for parallel association mining. In *Proceedings of the 4th Symposium on Knowledge Discovery and Data-Mining*, 1998.

[39] S. Parthasarathy, M. Zaki, M. Ogihara and S. Dwarkadas. Incremental and interactive sequence mining. ACM Conference on Information and Knowledge Management 1999.

[40] S. Parthasarathy, and M. Ogihara. Clustering Homogeneous Distributed Datasets.. Fourth Practical applications of Knowledge Discovery and Data Mining (PKDD), 2000.

[41] S.Parthasarathy, S. Dwarkadas, and M. Ogihara. Active Mining in a Distributed Setting. In *Workshop on Parallel and Distributed KDD systems*, 1999.

[42] E. Pinheiro, D. Chen, S. Dwarkadas, S. Parthasarathy, and M. L. Scott. S-DSM for Heterogeneous Machine Architectures. In *Second Workshop on Software Distributed Shared Memory*, Santa Fe, NM, May 2000.

[43] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. In Third International Workshop on Discrete Algorithms and Methods in Mobile Computing and Communications, Seattle, WA, Aug. 1999.

[44] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[45] M. Shapiro, S. Kloosterman, and F. Riccardi. PerDiS — A Persistent Distributed Store for Cooperative Applications. In *Proceedings of the Third Cabernet Plenary Workshop*, Rennes, France, April 1997.

[46] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in beehive. In *PROC of the 9TH SPAA*, June 1997.

[47] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining Temporal Coherency of Virtual Data Warehouses. In IEEE Real-Time Systems Symposium (RTSS98), Dec. 1998.

[48] D. Sriram, R. Logcher, N. Groleau, and J. Chernoff. Dice: An object oriented programming environment for cooperative engineering design. In *AI in Enginnering Design, Vol. 3, Academic Press*, 1992.

[49] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network In *Symposium on Operating Systems Principles*, October 1997.

[50] R. Subramonian and S. Parthasarathy. A framework for distributed data mining. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[51] D.B. Terry, M.M. Theimer, K. Peterson, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *15th SOSP*, Dec 1995.

[52] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. Incremental updation of association rules. In *KDD97*, Aug 1997.

[53] M. vanSteen, P. Homburg, and A.S. Tanenbaum. The architectural design of Globe: A wide-area distributed system. In *Technical Report (Vrije University) IR-431*, March 1997.

[54] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, Sept. 1992.

[55] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.

A Footnotes

* This is an expanded version of our conference paper in the First SIAM International Conference on Data Mining [36]. This work was supported in part by NSF grants CDA–9401142, EIA-9972881, CCR–9702466, CCR-9988361, and CCR–9705594; and an external research grant from Compaq.

1. Note that our implementation of a write lock is *relaxed*, in the sense that readers are permitted during a lock held in write mode.

2. Each complex-object is mapped to a disjoint set of pages, which enables our system to transparently detect changes to objects using virtual memory hooks. Since we are dealing with applications where the complex-objects are reasonably large relative to the size of a page, this does not result in memory wastage.

3. Note that since complex-objects can dynamically change in size, all the pages for a complex-object need not be contiguous, so a slightly modified form of array indexing is needed.

4. In order to guarantee an atomic update, the data cannot be sent in an as is condition, as partial changes to the object may have occurred concurrently.

5. Clients need to execute this only once. Subsequent updates from the server are automatically handled correctly by our system's address translation mechanisms.

6. Support is the number of times the pattern occurs in the dataset.

7. The datasets used are described in [41].

8. Going to larger window sizes did not affect the average synchronization times for our workloads.

9. Object techniques involving re-executing methods on all cached copies are not possible on data mining applications, since updates use large amounts of I/O, with the data residing only on the server. Hence, these approaches are not suitable for such applications.

B Figure Captions

- Figure 1: Interactive Client-Server Mining
- Figure 2: Server-Side Pseudo Code: *InterAct* method-calls are shown in **bold**
- Figure 3: Client-Side Pseudo Code: *InterAct* method-calls are shown in **bold**
- Figure 4: Efficient Shared Data Updates
- Table 1: Server Update Properties
- Table 2: Time (in seconds) to Execute Query: 100Mbps Interconnect
- Table 3: Time (in seconds) to Execute Query: 10Mbps Interconnect
- Figure 5: Coherence Model Evaluation
- Table 4: Association Performance Breakdown
- Table 5: Sequence Mining Performance Breakdown
- Table 6: Discretization Performance Breakdown
- Table 7: Similarity Performance Breakdown
- Figure 6: Effect of Client Controlled Coherence on Similarity Performance
- Figure 7: Result Quality
- Figure 8: Effect of Transaction Rate on Similarity Performance