Abstract

We propose an architecture for scalable persistent object managers that provide access to large numbers of objects distributed over a variety of physical media. Our approach is lightweight in that we are interested in providing direct support for the creation, access, and updating of persistent objects, but only indirect support for the other functions traditionally associated with an object-oriented database, such as transactions, backup, recovery, or a query language. This design allows application programmers access to the productivity and performance of using objects, while relying on an underlying hierarchical storage system to manage the large amounts of data.

Our design is layered and multilevel in that it caches and migrates large-grained physical collections of objects called folios from tape to networked disks. Separately, it also caches and migrates smaller-grained physical collections of objects called segments between nodes on a network. Segments are then moved into memory as usual for persistent object managers.

In this paper, we also describe the implementation of a system called PTool based upon this design and give a description of preliminary performance results. Previously, in version 0.4 of PTool, we used a single-level caching algorithm between the hierarchical storage system and the object manager. This algorithm is described in the Twelfth IEEE Symposium on Mass Storage Systems. PTool, with this caching algorithm, has been used in high energy physics and aeronautics. On the basis of this experience, a multilevel caching algorithm was designed and implemented in Version 0.6 of PTool. This version of PTool has been used for applications in high energy physics, aeronautics, decision support, and multimedia applications. Since then, in version 2, we have redesigned PTool to take advantage of specialized segment managers to handle networked disk, tape, and other media.

This research was supported in part by NASA grant NAG2-513, DOE grant DE-FG02-92ER25133, and NSF grants IRI 9224605 and CDA-9303433, The Laboratory for Advanced Computing at UIC, and by the National Scalable Cluster Project.

Introduction

A current challenge is to develop low overhead, high performance persistent object stores for scientific and engineering applications that scale as the amount of data increases and that balance the input-output demands and processing demands of applications. In this paper, we are specifically concerned with understanding some of the issues and strategies in systems which provide transparent access to large numbers of objects distributed over a variety of physical media.

The essential issue is the way in which physical collections of objects are accessed, cached, and migrated. Recall that an object is called persistent when it exists independently of the process that creates it. In other words, persistent objects have been written to permanent media, such as a disk. We assume that objects are gathered together into physical collections called segments, as is usual for persistent object managers, and that segments are themselves gathered together into physical collections called folios. This hierarchy is essential for scalable persistent object stores, since managing a terabyte sized object store using segments alone is impractical due simply to the number of segments required.

Our approach can be characterized as lightweight in the sense that we are interested in the design of a system that provides scalable access to collections of objects distributed over a variety of physical media, but we are not interested in providing direct support for many of the features traditionally found in an object-oriented database, such as transactions, backup and recovery, or a query language. Rather, when these features are needed, we feel that they should be added incrementally by interfacing additional components, as illustrated in Figure 1.

In this paper, we describe an architectural design, an implementation, and our early experience with a system...
Recent work has been growing interest in coupling databases to hierarchical storage systems. For example, implementations that interface a hierarchical storage system to a relational database have been described ([8],[9]). A proposal for extending POSTGRES to provide support for accessing tertiary storage was made by Stonebraker [10]. Access to data provided by NASA’s proposed Earth Observing System also require that databases be interfaced to hierarchical storage systems [11]. The challenges to the database community provided by trying to access tuples on tape is nicely summarized in [12].

The caching algorithm we propose here is analogous to caching algorithms used for distributed file systems [13] and [14]. There have been a variety of caching algorithms for client-server database systems that utilize a page-server architecture ([15],[16]). In these systems, a number of clients make requests to a server for pages. Our caching algorithm can be viewed as a multilevel analogy to these types of systems.

**Model**

Recall that our goal is to understand how to design and develop persistent object stores which scale as the amount of data increases and which balance the input-output and processing requirements of high performance applications. Also recall that our approach is lightweight in the sense that we are interested in providing direct support for creating, accessing, and updating collections of persistent objects, but provide only indirect support for functions such as transactions, backup, and recovery.

For the *logical* model, we assume that there are objects, that objects belong to collections, and that collections belong to stores. Collections themselves are objects and hence can belong to other collections. Objects can be referenced directly through an object ID or indirectly by requesting the next object in a collection. See Figure 2.

For the *physical* model, we assume that there are objects, that objects are gathered into physical units of fixed size called segments, and that segments are gathered into physical units of fixed size called folios. An object store physically consists of one or more folios. See Figures 2 and 3.

For the *architectural* model, we assume that processes make requests to persistent object managers for objects. Also, when required, we assume that persistent object managers make requests to segment managers for segments.
Figure 2. At a logical level, objects are grouped into collections and collections into object stores. Objects themselves can also belong to object stores directly. At a physical level, objects are grouped into segments, segments into folios, and folios into object stores. Note that objects and object stores have both a logical and physical existence.

Finally, when required, we assume that segment managers make requests to folio managers for folios. See Figure 4.

One variant of the architectural model is for the segment manager to keep track of the folios so that no separate folio manager is needed. Another variant is for the hierarchical storage system to manage the folios and for either the segment manager or the folio manager to request the desired folios from the hierarchical storage system.

Figure 4. This figure illustrates the interfaces between the components in the architectural model. The multi-level caching algorithm we propose here is based upon caching and migration of different granularities of objects between different levels in this architecture: folios between the storage system and the folio manager and segments between the folio manager and segment manager. As illustrated, for some applications it is convenient for the segment manager to interface directly to the hierarchical storage system.

The reason for this hierarchy is easy to see. In a typical implementation, the persistent object manager moves segments from disk to memory as needed. Segments are analogous to blocks in file systems. Segments can vary in size from tens of kilobytes to megabytes. The problem is that a terabyte size store would require $10^6$ 64 kilobyte segments, which is impractical to manage. For this reason, segments are gathered into folios so that the folios can be managed separately.

Architectural design

Components

As mentioned previously, to specify the physical location of an object requires specifying the store that contains it, the correct folio within the store, and the correct segment within the folio. Finally, the location is determined by specifying the offset within the segment. The architectural model provides separate components to manage objects, segments, folios, and storage.

1. **Persistent object manager.** The persistent object manager itself creates and accesses persistent objects. It also creates, opens, and closes stores.
2. **Segment manager.** If the segment containing a referenced persistent object is not currently available in memory or virtual memory, the persistent object
manager generates a fault to the segment manager. If the segment is currently maintained by the segment manager, it is returned to the persistent object manager; otherwise, it is requested from the folio manager. A segment manager serves one or more persistent object managers. The persistent object manager may also stripe segments across several segment managers.

(3) **Folio manager.** If the segment required by the segment manager is not currently maintained by it, then the segment manager generates a fault to the folio manager. The folio manager then determines the location of the folio containing the required segment, retrieves the folio, extracts the segment, and returns the segment to the segment manager. The folio manager may interface to the hierarchical storage system or maintain the folios directly. A folio manager serves one or more segment managers. The segment object manager may also stripe folios across several folio managers.

(4) **Hierarchical storage system.** This design allows for either the hierarchical storage system itself to function as the folio manager, or alternatively, for the folio manager to interface to the hierarchical storage system.

**Interfaces**

The essential issue of concern in this paper is how physical collections of objects are accessed, cached, and migrated. Since hierarchical storage systems provide analogous support for files, it is important to understand the best way to interface persistent object managers and hierarchical storage systems. There are several approaches, depending upon whether the responsibility for determining the physical location of a folio is up to the object store or the hierarchical storage system.

(1) Implement the folios as files, layer the object store over the hierarchical storage system, and let the hierarchical storage system manage the folios. One could either use the hierarchical storage system as a surrogate for the folio manager or interface the hierarchical storage system to the folio manager. Note that in this case the decision of whether a folio is migrated to tape is left up to the hierarchical storage system.

(2) A closely related alternative is for the hierarchical storage system to manage folios as raw storage using a storage system interface instead of a file system interface. In this case there would probably be no need for a separate folio manager whose role would be assumed by the hierarchical storage system. Also, note that in this case the decision of whether a folio is migrated to tape is left up to the hierarchical storage system.

(3) Implement the folios as files, but assume that the object store itself tracks the physical location of the folios and has separate interfaces to disk, tape, and other physical media.

**Prefetching and striping**

In practice, we found that one of the easiest and most effective means to improve performance in a persistent object manager is by prefetching segments and folios. We found that the actual prefetching strategy was of less importance than the fact that it was used at all. Prefetching is also an easy way to introduce a certain amount of parallelism. For example, if segments \( n + 1, n + 2, \ldots \) are prefetched when segment \( n \) is accessed, then this can be done in parallel if these segments are physically striped across several disks.

**Multilevel caching (MLC)**

This section is adapted from [17]. In this section, we describe a multilevel caching algorithm for a persistent object store on a distributed collection of nodes. We assume that each node has a local disk.

The role of the segment manager is to obtain segments from the folio manager and to cache and migrate the acquired segments. Segment managers run on each node and manage a cache of segments stored on the node's local disk. Segments are moved into the processor's virtual memory when an object is referenced.

When the segment buffer is full, a standard least recently used (LRU) criterion determines the segment to be replaced. If the segment is dirty, it is returned to the folio manager; otherwise the folio manager is notified that the segment is no longer in use. In other words, a variant of write on close is employed. To control the consistency of the segments, the segment manager supports sequential write sharing: several clients can open a segment for reading, but only one client at a time has write permission. In principle, this approach does not scale well; in practice, since our applications tend to be read often, write rarely, it has not yet presented a problem.

When a reference to an object causes the segment manager to request a segment from the folio manager, the segment manager also prefetches the following four segments. This is done in parallel and pre-emptively: if any segment other than those requested in the prefetch is requested by the segment manager, the prefetch request is aborted and the prefetch corresponding to the requested segment is initiated.

Figure 5 describes this algorithm in more detail.

**Implementation**

This section is adapted from [2]. To study our multilevel caching algorithm, we used a scalable persistent object...
The multilevel caching algorithm is summarized in this figure. Both the segment cache manager and the folio cache manager use a LRU replacement policy when the cache is full, which we have not included in order to simplify the presentation.

A particularly simple implementation of PTool gathers segments into folios, implements folios as files, and names folios by prefixing the store name to folio number. For example, folio number 428 in the object store named “EventDB” would be a file named “EventDB.428.”

In this implementation, the majority of the functions in PTool are concerned with the allocation and access of objects in segments. A segment for PTool is simply a block of contiguous memory that can be placed on and removed from a physical storage device. PTool functions that deal with segments are of one of two types:

1. **Device independent functions.** The majority of segment related functions are of this type. Examples include functions for allocating objects within segments and for storing and retrieving segments.
2. **Device-dependent functions.** For example, there are separate functions for working with segments on local disk, networked disks, and tape.

This provides a great deal of flexibility. An abstract base class contains the device-independent segment manager functions. Device-dependent functions in this class are virtual functions. In this way, PTool exploits a virtual segment manager. Several segment managers are created that can be invoked. At run time, the store is examined and the correct virtual segment manager is created for the appropriate type of store and media.

For example, the declaration

```
Stores("EventDB", Store:ReadOnly, Store:Remote)
```

indicates the persistent object store named “EventDB” is read only and stored on a remote (networked) disk. Because each segment manager is so task specific, it can be simple
Fourteenth IEEE Symposium on Mass Storage Systems

Figure 6. A large portion of the PTool functionality resides in the segment manager. The device-independent functions are part of the virtual segment manager and the segment manager base class. Device-dependent functions are part of specialized device-dependent segment managers, which are quite small. For this paper, a specialized segment manager was developed for tape.

Experimental results

In this section, we summarize some of the experimental work involving PTool.

In the first subsection, we described measurements of the effectiveness of the MLC caching algorithm as the number of segments increases for sequential access to data on a single node system. With the parameters chosen, one expects a theoretical speedup of 50 percent. It turns out that this is approached as the number of segments increases. For random access to data, the overhead of prefetching can degrade performance. This is measured in a second series of tests. It turns out that prefetching is more important for multi-node systems.

In the second subsection, we summarize a system developed using PTool that accessed approximately 40 gigabytes of experimental data on disk and a terabyte of data on tape. The third subsection describes a variant of this system using random data of a similar format that was easier to instrument.

Multilevel caching (MLC)

This section is adapted from [17]. As mentioned before, the multilevel caching algorithm was implemented in PTool version 0.6. Some of the tests we ran with this implementation are summarized in Table 1. Prefetching segments in optimistic cases results in a 40–50 percent speedup. In worst cases, prefetching can increase the cost of queries. Pre-emptively aborting prefetches puts an observed upper bound of around 15–20 percent in this increase. Running a set of application specific benchmarks for high energy physics data resulted in a performance speedup of around 30 percent.

The top table illustrates the time in seconds for queries which compute with the objects in sequential order. Ideally, the MLC algorithm should be twice as fast, with a ratio of 0.5, as the naïve algorithm, since the naïve algorithm only requests a new segment after finishing the computation with the current segment. As the number of segments used in the query increases, this ratio is approached.

The bottom table illustrates the time in seconds for queries which compute with segments in arbitrary order. Ideally, the MLC algorithm should be no slower than the naïve algorithm with a ratio of 1.0. This holds approximately.

Prefetching segments in optimistic cases results in a 40–50 percent speedup. In worst cases, prefetching can increase the cost of queries. Pre-emptively aborting prefetches puts an observed upper bound of around 15–20 percent in this increase. Running our application specific benchmarks resulted in a performance speedup of around 30 percent.

These tests used: a store size of 16 MBytes; a segment size of 512 KBytes; a folio size of 8 MBytes; a segment cache in the segment cache manager holding 16 segments; and a persistent object manager that used 4 segment slots. The time to transport a segment over the network was 0.5 seconds; the time for a query was 0.5 seconds. Note that we test under the assumption that I/O and CPU demands are balanced. All times are in seconds.

FNAL eventstore

This section is adapted from [18]. In a joint project with scientists from the PASS Project at Argonne National Laboratory and the CAF Project at Fermi National Accelerator
Event data studies on an ATM cluster

MBytes of memory and were running AIX Version 3.2.5. Performance.

nodes in the IBM SP-2 is an effective means of improving both disk and tape. A careful performance analysis of transparently access data on disk and tape, accessing data on in its original legacy data format. Although queries could talcen from this paper.

In this system, it is possible to transparently query gigabyte-sized stores of event data on the IBM SP-2 spanning both disk and tape. A careful performance analysis of this system is currently in preparation. From a preliminary analysis [18], it appears that prefetching segments across nodes in the IBM SP-2 is an effective means of improving performance.

Event data studies on an ATM cluster

This section describes four studies of scientific data analysis using PTool. All of the studies used randomly generated data of a similar format to the actual experimental event data from the FNAL detector. The first two studies are taken from [19]. The latter two were done specifically for this paper.

Experiments were conducted on a cluster of four IBM RS/6000 Model 370 workstations connected by a FORE Systems ASX-2000 ATM switch. The workstations had 32 MBytes of memory and were running AIX Version 3.2.5. Ten gigabytes of SCSI disk were attached to each node.

In the first study, we compared the analysis of event data using PTool to the analysis of the data using functions which read and wrote tab-separated ASCII data. Our goal was to try to quantify the performance advantages of object-based vs. file-based access to data for analysis patterns similar to those arising in scientific computing. To do this, we created two data sets containing events: one using PTool and one using routines which read and wrote tab-separated ASCII data. In our model query, we fixed an attribute, and for each event in the data set, accessed this attribute. PTool was significantly faster than file-based access to the data for this model query. See Table 2.

The second study was concerned with the scalability of data analysis using PTool. For this study, we used PTool to populate disk-based event stores varying in size from 400 MBytes to 20 gigabytes. Time to query the data used the model query just described and it varied linearly with the amount of data.

The third study was concerned with the ability to access data from PTool using tape. For this study, a specialized segment manager was developed for tape. This was done by using most of the functionality from a general segment manager and adding a few specialized tape dependent access functions.

Our approach required that a type be associated with each store. Current types include local disk, remote networked disk, and tape. An attribute of each store specified its type and the appropriate segment manager was automatically invoked at run time. In this study, we populated and accessed a gigabyte size store of event data on tape. We emphasize that this approach supports the caching and migrating of stores from tape to disk and back. It demonstrates that it is relatively easy for lightweight persistent object managers to manage stores on a variety of media when necessary. In practice, we assume that this functionality though would often be assumed by the hierarchical storage system.

In the fourth study, we examined the speedup gained by striping disks with event data. For this study, we developed another specialized segment manager supporting disk striping. We showed that striping improved performance linearly with the number of nodes, from one to four nodes, as illustrated in Table 5.
In this paper, we described an architectural design for a scalable, multilevel persistent object store. Our design is targeted at scientific and engineering applications which require low overhead, high performance access to large amounts of data. As in the test above, each event object was accessed and PTool was used to dereference an attribute from it. This table is from [18].

<table>
<thead>
<tr>
<th>number of events</th>
<th>2 million</th>
<th>20 million</th>
<th>60 million</th>
<th>100 million</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage size (GBs)</td>
<td>0.413</td>
<td>4.13</td>
<td>12.54</td>
<td>20.91</td>
</tr>
<tr>
<td>user time (sec)</td>
<td>21.280</td>
<td>216.0</td>
<td>644.0</td>
<td>1118.0</td>
</tr>
<tr>
<td>system time (sec)</td>
<td>18.610</td>
<td>226.0</td>
<td>647.0</td>
<td>1062.0</td>
</tr>
<tr>
<td>wall time (min:sec)</td>
<td>1:24</td>
<td>13:48</td>
<td>41.12</td>
<td>66.41</td>
</tr>
</tbody>
</table>

Table 4. Accessing tape-based event objects using PTool. In this study, we used a segment manager specialized for object stores residing on tape. We created a 1.0 gigabyte object store of events and ran the same query as describe above. The query was I/O limited.

<table>
<thead>
<tr>
<th>storage size (GBs)</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>user time (sec)</td>
<td>3.240</td>
</tr>
<tr>
<td>system time (sec)</td>
<td>14.620</td>
</tr>
<tr>
<td>wall time (min:sec)</td>
<td>63.56</td>
</tr>
</tbody>
</table>

Conclusion

In this paper, we described an architectural design for a scalable, multilevel persistent object store. Our design is targeted at scientific and engineering applications which require low overhead, high performance access to large amounts of data distributed over a variety of physical media. Our design is lightweight in that direct support is providing for creating, accessing, and updating persistent objects, but only indirect support for transactions, backup, recovery, and the other functions associated with an object oriented database.

Our design is layered and multilevel in that it caches and migrates large-grained physical collections of objects called folios from tape to networked disks. Separately, it also caches and migrates smaller-grained physical collections of objects called segments between nodes on a network. Segments are then moved into memory as usual for persistent object managers.

In this paper, we also describe the implementation of a system called PTool based upon this design and give preliminary performance results. These results indicate that PTool can provide lightweight object management for persistent object stores that are tens to hundreds of gigabytes in size. We plan to test a terabyte size persistent object store shortly.

References


Fourteenth IEEE Symposium on Mass Storage Systems


