

ObjectStore®

Technical White Paper

Designing C++ Applications for Scalability and Performance

ObjectStore
A Division of Progress Software Corporation
14 Oak Park
Bedford, Massachusetts 01730
Phone: +1-781-280-4000
www.objectstore.net

Copyright © 1989-2002 Progress Software Corporation. All rights reserved. ObjectStore is a registered trademark, and Real Time Event Engine and RTEE are trademarks, of Progress Software Corporation in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.

November 2002

Contents

Introduction	1
ObjectStore's Architecture	2
Single-Level Store	2
Page as the Unit of Transfer	3
How Data Is Transferred to the Cache	3
Architecture-Based Performance Goals	3
How to Maximize Performance	4
1. Increase Cache Size	4
2. Cluster! Cluster! Cluster!	5
What Is Clustering?	5
Strategies for Clustering	5
Which Data <i>Not</i> to Cluster	6
Partitioning as Large-Grained Clustering	6
3. Reduce the Size of the Working Set	8
Splitting a Class into Head and Body	8
Eliminating "Holes" in Classes and Structs	9
4. Choose an Appropriate String Representation	10
Fixed-Length Strings	10
Symbol Tables	10
String Pools	11
5. Choose an Appropriate Collection Type	12
Do You Need a Collection?	12
Index-Only Collections and Dictionaries	12
Sets vs. Lists	12
Dynamic Extents	13
Pros and Cons of Presizing a Collection	13
6. Optimize Access to Large Collections	14
Adding an Index to Optimize Queries	14
Ordering Iterations by Address	14
Conclusion	15

Introduction

One of the major performance issues facing many distributed database applications is the question of scalability. Can you increase the size of the database or add more clients to the application without degrading performance?

Consider a database application consisting of several distributed clients that communicate across a network with a database server. (Note that, as used in this white paper, the term *client* can also refer to an *application server* in a middle-tier environment.) The server and each of the clients are on separate machines. When the application first goes into production, its performance is acceptable. But as you add new clients, the overall performance — not just that of the new clients — begins to degrade. And the more clients you add, the slower the performance.

The problem is that the application does not scale. The server can only handle a limited number of client requests. Each additional client brings with it a battery of data requests that, when combined with the requests of the other clients, overwhelms the server. The server cannot keep up with the increased rate of the requests and has become a bottleneck that slows down the entire application. This pattern is typical of applications that cannot scale to meet the demands of added clients.

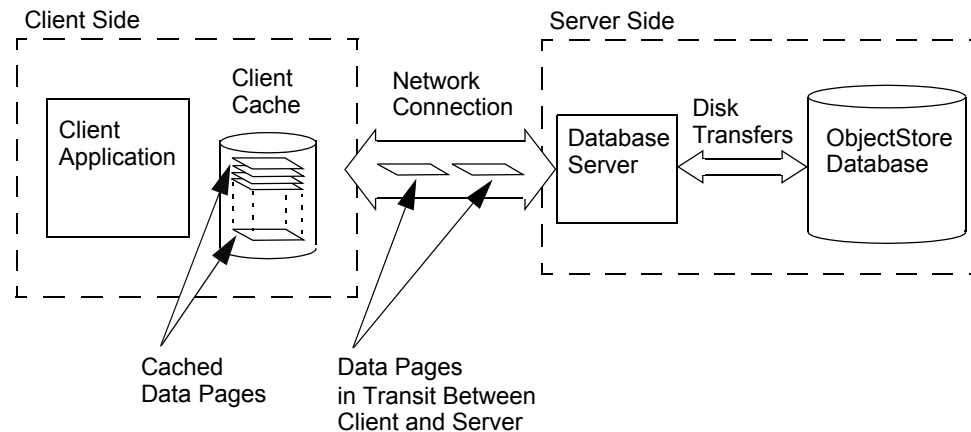
For such applications, scalability is handled instead by upgrading the hardware — for example, by replacing the server host with a more powerful Symmetric Multiprocessor (SMP) server or by adding smaller clustered servers. One problem with this approach is that the additional hardware and the associated maintenance costs are expensive. Also, depending on the number of clients that are added to the system, the additional hardware might not be a long-term solution. The application might have to scale to the point where it exceeds the capacity of existing and affordable hardware.

For applications that use ObjectStore®, scalability is achieved not by upgrading the hardware but by designing the application to take full advantage of ObjectStore's architecture. Processing that on a server-centric database management system (DBMS) is performed solely on the server side is distributed by ObjectStore's Cache-Forward™ architecture across each of the client applications. This client-centric design frees up the ObjectStore database server, allowing the addition of a nearly unlimited number of clients without straining the server. In an ObjectStore application, most of the work is divided among the clients, making it a truly distributed database application.

This white paper discusses how to maximize the performance of ObjectStore applications by taking advantage of ObjectStore's unique architecture. The first part describes how the architecture works, focusing on those components that applications can exploit for performance. The second part discusses different techniques for taking advantage of the architecture.

ObjectStore's Architecture

ObjectStore applications typically consist of multiple client applications and database servers. For the sake of illustration, however, the following figure depicts a simple application consisting of a single client and a single server. Note, also, that clients and servers can run either on the same machine and communicate by means of shared memory mapping, or on separate machines that communicate through a network protocol (as illustrated here).



You can refer to this figure in the discussion of the architecture in the following sections.

Single-Level Store

ObjectStore uses a *single-level store* for storing persistent data — that is, ObjectStore uses the same layout for storing data in a database that the data has in program memory. The on-disk format directly corresponds to the in-memory format. Any object that can be described in the C++ or Java programming language can be stored by ObjectStore.

Furthermore, the collocation of data — the locality of data relative to other data — is the same in the ObjectStore database as it is in program memory. For example, when an application uses the placement syntax of the C++ `new` operator to collocate storage for related objects, the on-disk and in-memory locality of those objects are the same.

ObjectStore's single-level store has two important implications:

- No translation occurs when data is transferred between program memory and the database. Data is transferred between the program and the database directly, without the need for mapping code to reformat data from its in-memory representation to its on-disk representation, or vice versa.
- The application — not the database — controls how and where data is laid out in the database. As will be seen, this control over the data gives the application control over ObjectStore's performance.

Page as the Unit of Transfer

Although the client application accesses data at the object level, data is transferred between the database server and the client application at the minimum granularity of a page. Once a page is transferred to the client cache, the page is transparently mapped into the application's virtual address space and all data on the page becomes accessible to the local client applications.

Page size is determined by the operating system of the database server's host machine.

In general, transferring a page is more efficient than transferring an object because a page can contain more than one object, and applications generally operate on more than one object at a time. Also, transferring pages rather than objects allows ObjectStore to take advantage of the fact that operating systems and networks are optimized to handle fixed-size blocks of data.

Although ObjectStore allows transfers that are multiples of a page, single-page transfers meet the data requirements of most applications.

How Data Is Transferred to the Cache

A data transfer is initiated when a client application attempts to dereference a pointer to persistent data on a page that is not cached. This attempt to dereference triggers a page fault. ObjectStore traps the page fault and then executes the following sequence automatically and transparently to the application:

1. A request for the data is sent from the client to the server (across the network if the server is on a different machine).
2. The server performs a disk access to fetch the page with the requested data, or the data is retrieved from a server memory buffer.
3. The page is sent across the network from the server to the client.
4. The page is made resident in the cache and mapped to the client application's address space.

This cycle occurs for each request for uncached data. If the cache is full, ObjectStore evicts the least recently used page to make room for a new page. If a page containing requested data is already in the cache, there is no network traversal, no communication with the server, and no disk I/O — and no performance degradation that results from these activities.

Architecture-Based Performance Goals

How, then, does an application make use of ObjectStore's architecture for optimum performance? It does so by achieving two goals:

- Minimizing the number of pages transferred between server and client
- Maximizing the use of pages already in the cache

Both goals are related. The more the application uses pages in the cache, the fewer the pages that have to be transferred from the server. The fewer pages that are transferred from the server, the

more likely that all of the transferred pages will fit together in the cache. And, finally, both goals have the underlying purpose of reducing the overhead of network traffic and disk I/O.

How to Maximize Performance

On many DBMSs, performance tuning takes the form of setting different server-based parameters. Such parameters are also available for tuning ObjectStore's server. However, because of ObjectStore's client-centric architecture, the most effective way to maximize performance and scalability is to design the application to take full advantage of the client cache.

Accordingly, the performance techniques described in the following sections show you how to make the most effective use of ObjectStore's Cache-Forward architecture. Some of the techniques are fairly simple to implement; for example, the technique described in the first section ("1. Increase Cache Size." on page 4) requires no code change but can improve performance dramatically. Other techniques might require editing source files, and some are more appropriate to implement during the design phase.

None of these techniques should be taken as representing rules for using ObjectStore. Nor do they exhaust the possible ways to maximize performance on ObjectStore. Rather, they are intended to suggest different approaches to improving performance when viewed from the perspective of the client.

1. Increase Cache Size.

The best way to make optimum use of the client cache is to make sure that its size is adequate for the application's *working set*. The working set consists of all the persistent data that is accessed by a top-level operation in the application. Ideally, the cache should be large enough to hold the largest of the application's working sets. The more of the working set that can fit in the cache, the fewer the pages that must be transferred from the server and the better the performance.

The size of the cache should be set for each application, based on the sizes of the working sets accessed by the application. For example, an application that accesses data that is randomly scattered across many pages will require a larger cache than an application that accesses well-clustered data; see "2. Cluster! Cluster! Cluster!" on page 5. In general, the cache size should be large enough to fit the largest working set in the application. Too small a cache can result in increased network and disk I/O activity.

The maximum limit for the cache size will depend on the load on the host machine. If many applications are running concurrently on the same machine, too large a cache can result in poor performance because of thrashing — the cache uses up so much memory that there is not enough for other applications to run without increased page swapping. If you need a larger cache than your machine can support, you should consider either reducing the amount of data that the application accesses ("3. Reduce the Size of the Working Set." on page 8) or partitioning the data set (see "Partitioning as Large-Grained Clustering" on page 6).

2. Cluster! Cluster! Cluster!

The single most important way to maximize performance is to cluster the data in the working set. If the data is well clustered, only the minimum number of pages needs to be transferred to the cache. Once in the cache, the pages can be fully utilized by the application, without any additional page transfers.

The following sections describe how to cluster data effectively.

What Is Clustering?

Clustering refers to the arrangement of persistent data on the disk so that data that is accessed together is grouped together. Instead of spreading data randomly throughout the database, clustering concentrates it in dense groups, where each group represents a working set. The idea is that when the server transfers a page to the client, the page will include not just the object that the client requested but also the objects that it is going to need. Clustering is a way to anticipate which data the application will need.

An ObjectStore database is organized into *clusters* and *segments*. The segment provides access controls on the data managed by the segment and allows for the partitioning of data (see “Partitioning as Large-Grained Clustering” on page 6). A segment consists of clusters, which refer to physical locations within an ObjectStore database. The cluster is the basic unit of allocation.

Individual segments and clusters are accessible by means of the `os_segment` and `os_cluster` classes in the ObjectStore class library. For detailed information about these classes as well as the API for allocating and clustering persistent data, see the ObjectStore C++ *API Reference* and the C++ *API User Guide*.

Strategies for Clustering

Here are three common strategies for clustering data:

- By parent — Cluster the child object with its parent object.
- By type — Cluster objects of the same type together.
- By attribute value — Cluster objects of the same class in different groups according to the value of a class attribute.

The following paragraphs discuss each strategy.

Clustering by parent. This is the most common strategy for clustering and is appropriate when the application accesses one object (the parent) in order to access another (the child). In this situation, the child should be clustered with its parent.

To take a simple example, consider the following definition of the `email` class:

```
class email { //
private:
    const char* address;
    const int date;
    const char* subject;
    const char* message;
public:
```

How to Maximize Performance

```
    email(char* addr, int d, char* subj, char* msg);  
};
```

If `subject` is always accessed with its parent (`email`) and both are in the same working set, then both should be clustered together.

Other examples include clustering hierarchically related objects (for example, `car` and `tire`), where the application typically does not access the child (`tire`) without also previously accessing the parent (`car`). Likewise, objects in a tree should be allocated so that the nodes (children) are always clustered with the root (parent).

Clustering by type. If all instances of a class are typically accessed together, then they should all be clustered together. If there are many instances, it might not be possible to fit them all in the cache. In this case, you should consider whether you need to access all data members in the class and exclude those you do not need from the working set; see “Splitting a Class into Head and Body” on page 8.

Clustering by attribute. If a limited subset of the instances of a class extent are in the same working extent, only instances in the subset should be clustered together. One way to define a subset is by the value of a class attribute so that all instances that have the same value are in the same cluster. In the `email` example, all instances of `email` for the same year (as encoded in the `date` data member) could be clustered together. To keep track of which year goes in which cluster, you can use a persistently stored lookup table with year-cluster entries; given a year, the table would return a pointer to the cluster with the `email` instances for that year.

Which Data *Not* to Cluster

Clustering is as much a matter of knowing which data to exclude from the cluster as which data to include. In response to a client request for data, the server transfers a page that includes both requested and *unrequested* data. If the database is well clustered, the unrequested data will consist of data that the application will *soon* need — that is, other data in the working set.

But the requested page might also include data that is not part of the working set and is therefore unneeded. Transferring pages that include a high percentage of *unneeded* data can be inefficient if it takes up space that could have been allocated for *needed* data or if it results in more pages being transferred than are strictly needed to hold the working set.

The placement syntax of the C++ `new` operator can be used to ensure that unneeded data is not clustered with needed data.

Partitioning as Large-Grained Clustering

So far, this white paper has presented scalability as becoming an issue only when more clients are added to the application. But it can also become an issue when the size of the database increases. The data set can grow to the point where it becomes unmanageable — for example, the working set becomes too large to fit in the cache — unless the application and the database are designed to handle it.

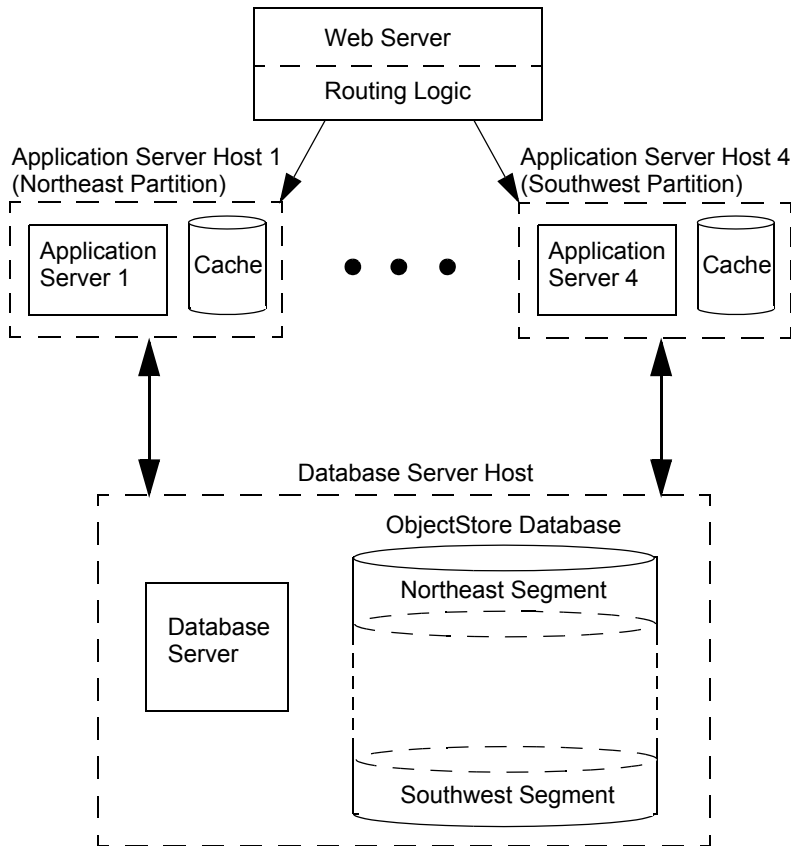
As mentioned in “What Is Clustering?” on page 5, an ObjectStore database is organized into segments, and each segment into one or more clusters. The segment provides one way to scale the application with respect to the size of the database. Using segments, you can divide the data set into logical *partitions* and assign each partition to a segment. Depending on the size of the data set, instead of assigning the partitions to segments you can assign them to clusters or databases.

But this discussion will assume the use of segments for partitioning.

Consider as an example a phone directory application, in which an application server processes requests from one or more Web servers for directory information. Suppose also that the database is expected to grow to 2 GB in size. Here is one way to use partitions to scale the application:

- Partition the data into four geographical districts — for example, northeast, southwest, and so on.
- Cluster the data in each partition into one of four database segments.
- Add three more application servers — each with its own 500 MB cache — so that each partition can be assigned to one of the four application servers.
- Include routing logic — such as HTTP request redirection in a Web servlet or CORBA object request routing — in the Web server so that it can route incoming requests from Web browsers to the appropriate application server.

The following illustration depicts the example application:



If the data set in this example is fully clustered, and the data and processing are partitioned evenly across the four client applications, then the entire database can be cached across the four clients, with little or no disk or network activity beyond what is required to fill the caches. Once the caches are loaded, the application can run at in-memory speed, as there is no need to constantly check with the database server for updates.

Note that, as described here, partitioning is a performance-tuning technique; it does not impose requirements on how requests must be routed. For example, after partitioning you might find that more load balancing is needed, that one application server is processing many more requests than another. To distribute the load more evenly, you can change the Web server to route some of the requests for data from one partition to a different application server. These requests will be processed correctly, as though they had been routed to the application server originally assigned to process them. ObjectStore ensures transactional correctness regardless of which cache gets data from which partition.

3. Reduce the Size of the Working Set.

To fit the working set in the cache, you can do either of two things: increase the size of the cache or reduce the size of the working set. There is a limit to how much you can increase the size of the cache without lowering performance through increased page swapping; see “1. Increase Cache Size.” on page 4. Before reaching this limit, you should consider reducing the size of the working set so as to fit it in the cache.

This section discusses the following ways to reduce the size of the working set:

- Splitting a class into two smaller classes
- Laying out data members in classes and structures to avoid “holes”

For additional information about reducing the size of working sets, see the following sections:

- “4. Choose an Appropriate String Representation.” on page 10
- “5. Choose an Appropriate Collection Type.” on page 12

Splitting a Class into Head and Body

This technique for reducing the size of the working set is useful when an operation accesses a large class extent but requires access only to some of the data members. This technique involves splitting the class into two smaller classes — the head and the body — and containing the required data members in the head and the others in the body. The head can also include a pointer to the body. The accessing operation scans the head instead of the original, larger class. The result is that more objects can be clustered on a page and fewer pages are required to bring the working set into the cache.

For example, consider an operation that scans instances of the `email` class listed in “Strategies for Clustering” on page 5. If the operation requires access only to the `address` and `subject` data members, these can be placed in the head class, and the unneeded data (`date` and `message`) can be placed in the body. Assuming that a `message` string averages 500 bytes in size, the reduction in the working set is considerable and, if the head objects are well clustered, can result in a significant reduction in the number of page transfers.

Note that in this example you can achieve nearly the same effect without having to split the `email` class, by doing the following:

- Allocate all data members except `message` with the `email` object in one cluster.
- Allocate the `message` strings in a separate cluster.

Eliminating “Holes” in Classes and Structs

As described in “Single-Level Store” on page 2, the on-disk layout of objects in the database directly corresponds to their layout in program memory. Consequently, if an application’s definition of a class or structure results in unused or inaccessible space in the in-memory layout of an object, the wasted space will also show up in the on-disk layout.

One source of wasted space is the padding (also known as *holes*) that the compiler inserts between data members of a class or structure to ensure correct data alignment. Padding increases the size of the object. Eliminating the padding helps to reduce the size of the working set.

Consider the following structure:

```
struct bad_align {
    char c1; // 1 byte
    // 7 bytes of padding
    double d1; // 8 bytes
    int i; // 4 bytes
    // 4 bytes of padding
    double d2; // 8 bytes
    char c2, c3, c4; // 3 bytes
};
```

Depending on the layout rules used by the compiler and the target platform’s alignment rules, the order in which the members of this **struct** are declared could result in 11 bytes of padding. Although not significant in the case of a single **bad_align** object, the padding can add up if the working set consists of an array of **bad_align** elements, requiring additional pages and therefore more transfers between client and server.

The compiler also pads between objects to ensure correct alignment for each object. Given an array of **bad_align** structs, for example, the compiler would add 5 bytes of padding between array elements to ensure 8-byte alignment for each **bad_align** element. (The alignment of a **struct** or **class** is determined by the alignment of its most demanding data member — in the case of **bad_align**, the most demanding member is a **double**.)

This additional padding means that the overhead of storing each **bad_align** element in the array is 16 bytes of wasted space. If the server had to transfer a 1000-element array to a client cache, the transfer would require 10 pages — 4 more than required without the padding.

To minimize or eliminate holes, data members should be arranged in descending order of alignment, starting with the most demanding, as in the following declaration:

```
struct good_align {
    double d1, d2; // 16 bytes
    int i; // 4 bytes
    char c1, c2, c3, c4; // 4 bytes
};
```

By having all of its data members declared to align on natural boundaries, this example requires no padding.

4. Choose an Appropriate String Representation.

Strings often account for a large percentage of the data stored in a database and are therefore a good place to look for ways to improve performance. Depending on how they are represented, strings can impact the size of the working set, locality of reference, and allocation cost.

Strings are frequently represented as variable-length strings, but they can also be represented as

- Fixed-length strings
- Strings stored in a symbol table
- Class-based string pools

The variable-length string is a popular representation because the C++ `new[]` operator makes it easy to allocate a string of exactly the right size, and the `delete[]` operator makes it easy to deallocate a string when no longer needed. However, you can sometimes improve performance by using other representations, as explained in the following sections.

Fixed-Length Strings

A fixed-length string is a character array of declared and fixed size, as in the following:

```
char state[3]; // 2 chars for the abbreviation, 1 for the null
```

Fixed-length strings have these benefits:

- The space overhead of a fixed-length string is only the length of the string. Compare variable-length strings, which also require four bytes for the pointer. If the variable-length string is wrapped in a string class, the overhead also includes metadata used by the string class.
- There is no allocation cost for fixed-length strings, as they are embedded in the parent object.
- If a fixed-length string is a member of a class, it is always collocated with its parent. This characteristic results in fixed-length strings always having good locality of reference. Compare variable-length strings, which are separately allocated. Even when the allocation for a variable-length string is clustered with its parent, if the string is later replaced with a different string, the replacement string might have to be allocated elsewhere and can end up on different page.

If performance is an issue, fixed-length strings can be used instead of variable-length strings whenever the length of the string is fixed or does not exceed 8–12 characters. If the string is much longer than this, the cost of potentially unused space starts to outweigh the benefits of the reduced overhead of fixed-length strings.

Symbol Tables

A symbol table can be used as a replacement for variable-length strings to eliminate duplicate strings and thereby reduce the size of the working set. You can implement it as a dictionary; see “Index-Only Collections and Dictionaries” on page 12. To use the symbol table, you would call a lookup function, passing the string that you want as an argument. If the string is not in the table, the function installs it in the table and returns a pointer to the newly installed string. If the string is in the table, the function returns its pointer.

To take advantage of ObjectStore’s architecture, the symbol table and its string contents can be

allocated in a dedicated cluster to ensure that they are closely clustered together.

A symbol table has the following benefits:

- Reduces the working set. Depending on the number of duplicate strings, a symbol table can significantly reduce the size of the working set. Consider a working set that consists of several hundred `employee` objects, with each object having one of four different `department_name` strings.
- Improves locality of reference. The symbol table and its list of strings can be clustered together to fit on 1–2 pages, enabling the entire table to stay in the cache.
- Reduces the cost of string allocation and deallocation. Only the unique strings require allocation. There is no deallocation because it is not needed. Effectively, using a symbol table replaces the cost of deallocating and reallocating duplicate strings with simple pointer assignments.

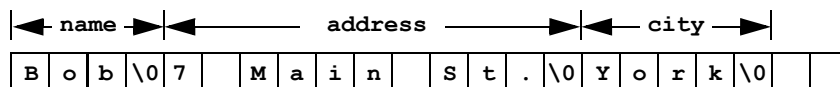
One drawback to a symbol table is that it can become a bottleneck in a concurrent application. Whenever a process writes a string to the table, it blocks other processes from reading the table. One way to avoid this problem is to create the strings ahead of time, when initializing the database.

String Pools

A string pool replaces all string data members in a class by concatenating them in one long string. Consider the following class:

```
class customer {
private:
    char* name;
    char* address;
    char* city;
    char* state;
    char* phone;
    . . .
};
```

To implement a string pool, you can add a `pooled_string` class as a data member of `customer` to allocate and manage the pooled string. Accessor functions (`set_name()`, `set_address()`, and so on) in the parent class would use the `pooled_string` class to perform their allocations and deallocations. The individual strings (`name`, and so on) would be assigned segments of the pooled string, as follows:



A string pool has these benefits:

- Increases the locality of reference by ensuring that all strings are on the same page and are clustered with their parent object.
- Reduces the number of allocations.

The disadvantage of a string pool is that it requires more work at run time to manage the string and to perform updates.

5. Choose an Appropriate Collection Type.

The ObjectStore collections facility provides a library of data structures for representing collections of objects. Your choice of a collection type will be guided mainly by your application's data model and access patterns. But you should also consider the performance implications of the different types and their features, as explained in the following sections.

Do You Need a Collection?

Before deciding to use a collection type, you should consider whether the number of objects and the access provided by the collection justify the overhead of the collection. The header for a collection type is at least 24 bytes in size, and an ordered dictionary that contains a small number of elements can be more than 1 KB in size. Consider a collection that is used as a subcomponent of a class. The overhead of the collection is now multiplied by the number of instances of the container class. If many of the collection subcomponents remain empty or are only partially filled, the wasted space can increase your application's paging overhead significantly.

Depending on the number of elements in the collection and how your application accesses the objects pointed to by the collection elements, a C++ array or a linked list might provide all the functionality that your application requires, without the added overhead of a collection.

Index-Only Collections and Dictionaries

Although adding an index can significantly increase the performance of query operations ("Adding an Index to Optimize Queries" on page 14), it can also increase the size of the database by adding another data structure — namely, the index.

You can, however, eliminate the overhead of the additional data structure associated with indexes and still get the benefit of indexing by using an index-only collection (`os_ixonly` and `os_Ixonly`). An index-only collection combines the functionality of an index and a collection in a single data structure.

Another alternative to an indexed collection is the dictionary (`os_Dictionary`). A dictionary can perform very fast `pick()` operations but, like an index-only collection, consists of only one data structure and therefore reduces the size of the working set. A dictionary uses a highly efficient data structure — a hash table for unordered dictionaries and a B-tree for ordered dictionaries — to perform lookup operations on its elements. Each element is associated with a key, which can be a value of any C++ fundamental or user-defined type. If the only reason for the collection is to perform lookup operations based on one key, the dictionary is a fast and efficient alternative to the indexed collection.

Sets vs. Lists

Among the types available in the collections facility are lists (`os_list` and `os_List`) and sets (`os_set` and `os_Set`). The main difference between them is that lists are ordered and sets are unordered. The ordered property of a list means that collection elements are inserted in specified positions and that the list maintains insertion order, as when you use a cursor to iterate over the list.

The ordered property of a list has a performance benefit not shared by sets. Because list elements are ordered, a list can have good locality of reference. When a list is allocated in its own cluster

and elements are added to the list in the same order as the objects they point to are created, the elements will be clustered in address order; see “Ordering Iterations by Address” on page 14. A list consisting of many elements will require a minimum number of pages to transfer and can easily fit in the cache.

A set, on the other hand, does not maintain insertion order. Elements are inserted in the list in random order, they have poor locality of reference, and the order in which set elements are visited in iterations is arbitrary.

One advantage of a set is that lookup operations are faster because a set uses a hash table to find the target element. Doing a lookup operation on a list, on the other hand, requires a linear search, which can take significantly longer. Also, inserting an element in a set is faster than in a list because a set does not need to insert the element in any particular order.

In general, however, in applications where good locality of reference is important, a list provides better performance than a set.

Dynamic Extents

If you need to use a collection as a class extent but do not wish to incur the overhead of storing the collection in persistent memory, you can use a *dynamic extent* (`os_dynamic_extent`). A dynamic extent can be allocated in either transient or persistent storage, as specified by the constructor. The advantage of a transiently allocated dynamic extent is that it is not stored in the database, and therefore the server transfers only the segment with the objects, and not the dynamic extent, to the client. The constructor allows you to specify a class and its extent as consisting of either all instances in the database or just the instances in a database segment.

A dynamic extent can perform many of the same operations of a persistent collection used as a class extent, including:

- Creating a cursor to iterate over the extent, including address-order cursors; see “Ordering Iterations by Address” on page 14.
- Performing queries.
- Adding an index to optimize queries; see “Adding an Index to Optimize Queries” on page 14.

Note, however, that a dynamic extent does not maintain cardinality and that the cost of computing cardinality manually is expensive, especially if done frequently.

Pros and Cons of Presizing a Collection

Collection types are extensible — that is, as you add more elements to a collection, ObjectStore dynamically allocates more space to provide slots for the new elements. However, these allocations can result in poor locality of reference because they are not always clustered on the same page as the rest of the collection.

You can prevent such suboptimal clustering by presizing the collection when you create it. The constructors for the collection types have a presizing argument that enables you to allocate a collection based on its expected number of elements. Presizing ensures that slots for newly added elements are clustered with the rest of the collection. Presizing has the added benefit of incurring the expense of allocating a collection at load time, when the performance cost of the allocation is less critical than when the collection is in use.

A word of caution about presizing: If you presize a collection for more elements than are actually added to it, the unused slots will unnecessarily increase the size of the working set; see “3. Reduce the Size of the Working Set.” on page 8. For a single collection, the unused space might not be an issue. But if the presized collection is a subcomponent of many objects, the unused space is multiplied by the number of objects and can increase the size of the working set and result in unnecessary and costly page transfers.

6. Optimize Access to Large Collections.

Querying and iterating over a large collection can be a lengthy operation, especially if the operation requires access to each member object. Even if you have clustered the objects associated with the collection, the iteration or query order might not correspond with the cluster order and therefore can result in an access pattern that skips from one page to another.

The following sections explain how to improve the performance of query and iteration operations.

Adding an Index to Optimize Queries

By default, a query operation scans all elements in a collection, visiting each page in the database that contains an element object. Depending on the number of elements in the collection and how widely scattered the element objects are throughout the database, a default query operation can be time-consuming.

Adding an index to a collection you plan to query can increase performance significantly. An index is a highly efficient data structure — a B-tree if the collection is ordered, a hash table if it is unordered — that minimizes the search for the target elements. Moreover, an indexed query does not access any of the element objects; it accesses only the index and the collection header.

It is important to keep in mind that an index is extensible; it *grows* dynamically as you add more elements to the indexed collection. To ensure good locality of reference for the index, you can create a cluster for exclusive use of the index. When you create the index, specify the newly created cluster in the argument list to `os_collection::add_index()`. ObjectStore will allocate initial storage for the index in this cluster and, as more elements are added to the collection, grow index entries in the same cluster.

Ordering Iterations by Address

An operation that iterates over a large collection and dereferences each object pointed to by a collection element can take much longer to perform if the size of the working set — that is, the collection object and the objects referenced by collection elements — exceeds the cache size. The obvious solution is to increase the size of the cache, as described in “1. Increase Cache Size.” on page 4. But this solution might not be possible if making the cache any larger results in bad performance because of increased page swapping.

This problem can occur whenever the access order of the operation does not match the order of the objects in memory. For example, if you use a default cursor to iterate over a collection, the iteration order is arbitrary. Although some pages might hold more than one object, the iteration is likely to skip around from page to page, without accessing all objects on one page before going on to the next page. As a result, a page might have to be evicted to allow room for another page, only to be refetched at a later point in the iteration.

You can solve this problem by creating a cursor that iterates over the objects in memory order. In the constructor for the cursor (`os_cursor` and `os_Cursor`), specify `order_by_address` as one of the arguments. This argument causes ObjectStore to sort the collection elements before the iteration begins, using the address order of the referenced objects. The sort ensures that objects are accessed in the same order in which they are stored in memory. Consequently, the operation will access all objects on a page before proceeding to the next.

Note that this solution does not deal with the problem of poorly clustered objects that are scattered across many pages. But it does ensure that, once a page is evicted during an `order_by_address` iteration, it will not have to be refetched in the same operation.

Conclusion

As explained in the first part of this paper, the focal point of ObjectStore's architecture is the cache, and the way to design scalable, high-performance applications is to take full advantage of the cache, using the techniques described in the preceding sections. Which of these techniques works best for your application will depend on its access patterns — that is, how your application uses its data. The key to good performance on ObjectStore, however, is to keep the cache full of the data your application needs, when it needs it. The sign of a well-designed ObjectStore application is that it is busy using data rather than requesting it.