Clustering Techniques

A Technical Whitepaper

By Lorinda Visnick

14 Oak Park
Bedford, MA 01730 USA
Phone: +1-781-280-4000
www.objectstore.net

Introduction

Performance of a database can be greatly impacted by the manner in which data is loaded. This fact is true regardless of when the data is loaded; whether loaded before the application(s) begin accessing the data, or concurrently while the application(s) are accessing the data. This paper will present various strategies for locating data as it is loaded into the database and detail the performance implications of those strategies.

Data Clustering, Working Sets, and Performance

With ObjectStore® access to persistent data can perform at in-memory speeds. In order to achieve in-memory speeds, one needs cache affinity. Cache affinity is the generic term that describes the degree to which data accessed within a program overlaps with data already retrieved on behalf of a previous request. Effective data clustering allows for better, if not optimal, cache affinity.

Data density is defined as the proportion of objects within a given storage block that are accessed by a client during some scope of activation. Clustering is a technique to achieve high data density. The working set is defined as the set of database pages a client needs at a given time. ObjectStore is a page-based architecture (see the *ObjectStore Architecture Introductory* technical paper at www.objectstore.net for a detailed review of the ObjectStore architecture) which performs best when the following goals are met:

- Minimize the number of pages transferred between the client and server
- Maximize the use of pages already in the cache

In order to achieve these goals, the working set of the application should be optimal. The way to achieve an optimal working set is via data clustering. With good data clustering more data can be accessed in fewer pages; thus a high data density rate is obtained. A higher data density results in a smaller working set as well as a better chance of cache affinity. A smaller working set results in fewer page transfers. The following sections in this paper will explain several clustering patterns/techniques for achieving better performance via cache affinity, higher data density and a smaller working set.

> NOTE: clustering is used in this paper as a concept of locality of reference. The term is not being used to refer to the physical storage unit available in ObjectStore. ObjectStore does present the user with a choice for location of allocations: with the database, within a particular segment, within a particular cluster. For the remainder of this paper, the discussion of cluster is a conceptual one, not the ObjectStore physical one.

Database Design Process

Database design is one of the most important steps in designing and implementing an ObjectStore application. The following steps are pre-requisites for a database design:

1) Identify key use cases (ones which need to be fast and/or are run frequently)

2) Identify the object(s) used by the use cases called out in step 1

3) Identify the object(s) that are read or updated during the use cases called out in step 1

The focus of clustering efforts should be on the database objects which are used in the high priority use cases identified above. Begin to cluster based on one use case, and then validate with others.

The database design strategies which lend themselves to achieving the optimal working set are:

- Clustering
- Partitioning

There are several different types of techniques which result in data being well clustered:

- Isolate Index
- Pooling
- Object Modeling

We will examine each of these in detail.

Data Clustering

Earlier in this paper we defined clustering as a technique to achieve high data density. Another definition of clustering is a grouping of objects together. If a use case requires objects A, B and C to operate, then those objects should be co-located for optimal data density. If upon loading the database, those objects are physically allocated close to one another, then we say we have clustered those objects. Assume that the size of the three objects combined is less than the size of a physical database page. The clustering leads to high data density because when we fetch the page with object A, we will also get objects B and C. In this particular case, we need just one page transfer to get all objects required for our use case.

To accomplish good clustering, one must know the use cases and the objects involved in those use cases. Given that knowledge, the goals of clustering are:

- Cluster objects together which are accessed together
- Separate (de-cluster, or partition – we will discuss partitioning in detail later in this paper) objects which are never accessed together. This includes separating frequently accessed data from rarely accessed data.

Note: A change of object model may be required to accomplish the clustering goals. To that end, we will examine some object model change techniques as part of our discussion.

Clustering Techniques: Isolate Index

An index on a collection yields faster query performance. If the index is placed, as it is by default, in the same physical location as the collection itself and the items in the collection, then poor locality of reference for the index items is likely to occur. Restated, all the items which are related to the index should be placed in close proximity to one another. All items related to the collection (excluding the index) should be placed in close proximity to one another. The index items should not be interspersed with the collection nor the items contained within the collection. Therefore, if we cluster the index and all the index items together in an area that is physically distinct from the collection and/or the items in the collection, the index will be faster to fetch. If there are no non-index items interspersed with the index items, then fewer pages will be required to fetch the index to perform a query or update the index.

Another “side” benefit of clustering index items in an isolated area is less fragmentation. Why? If the index were interspersed with the collection and items in the collection and then the index were dropped (most likely so that it could be regenerated) the physical locale where the index items had been would be fragmented. Fragmentation lowers data density. By definition, if you fetch a page and some space on that page is empty due to fragmentation, then you have lower data density.

Clustering Techniques: Object Pooling

Object pooling is a type of clustering that puts all instances of a certain type into one physical location. Imagine a scenario were an application expected to have 100 bank objects. An array, or pool, of 100 objects would be allocated at system startup. As bank objects were needed, a slot in the pool would be assigned for use by that particular bank. As banks are deleted, the slot is marked as available for reuse. This technique gives continuous storage for all objects of the type in the pool. Because deleted slots are reused, fragmentation is less likely. The lower fragmentation in combination with contiguous space leads to higher data density.

Clustering Techniques: Object Modeling

Object modeling is a technique that involves changes to the object model. In this category, we have four distinct methods:

- Head Body Split
- Data Member Ordering
- Collection Representations
- Virtual Keyword

Let us examine each of these in detail.

Head Body Split

Recall that in clustering not only do we want to put data together which is accessed together, but we want to separate data that is not accessed together. Some objects have atttributes which are key and other attributes which are rarely accessed. Such objects are candidates for the technique called Head Body Split. Consider a person object with all the following attributes:

- First name
- Last name
- Social security number
- Gender
- Eye color
- Hair color
- Height
- Weight
- Home address
- Home phone
- Cell phone
- Employer
- Work phone
- Work address

For most applications the only data that would be accessed from such a person object would be first name, last name and social security number. If all data were stored together, each person would consume a considerable amount of space. Then, a fetch of a database page would yield fewer people. Remember, one goal is fewer page transfers. Therefore, a larger person object is not desirable. However, a person is what it is – all data is needed at *some* time. This is a perfect situation for the head/body split. In the head portion the items that are frequently accessed are placed: first name, last name, social security number. The remainder of the attributes are placed in the body. The head has a pointer to the body. The heads are physically allocated in close proximity to one another; therefore a page fetch returns many heads. The bodies are allocated in a location physically apart from the heads. The end result is higher data density, fewer page transfers, and thus a performance gain.

Strings

Strings often represent a large percentage of a database. Therefore, an effective representation of strings can greatly reduce the database size. Smaller database size, fewer page transfers required to fetch data of interest. In addition to a smaller database size, some representations allow for fewer physical allocations. If allocations are reduced, then there is less overhead for the API calls for those allocations. In addition, an embedded string (one in which no separate allocation was required) has locality of reference to the parent object in which it resides. By definition, an embedded string is clustered with its parent object. Depending on the use cases for an application, it may be desirable to have embedded strings (clustered) or it may be desirable to have the strings allocated separately and placed in a different physical location (as with the head/body split).

In general, strings can fit into one of three categories:

- Variable-length String
- Fixed-length String
- Symbol table

Variable length strings are often represented via a "char *". As mentioned above, this will cause an additional allocation to occur for space to be assigned to the string. If you wish for locality of reference, then you must take special care to allocate the new string in close proximity with the rest of the object. By default, one is not guaranteed locality of reference when a separate allocation is performed. If using a third party package such as Rogue Wave (RW) or some other STL implementation, be aware that variable length strings often have a large amount of overhead associated with them. Larger objects mean fewer objects per page, which means more page transfers. In the case of RW, a variable length string has at least 24 bytes of overhead associated with it. Therefore, one should not consider using such an implementation thinking that they are saving space.

Fixed length strings can and should be used when locality of reference is desired. As mentioned above, an embedded string (char []) reduces the number of allocations and is guaranteed to be co-located with the rest of the object. If the fixed length string is large (for instance, the text of a paragraph or the list of ingredients on a label) then one might choose to use a char * representation (even though the string is of fixed length) instead of embedded string to achieve a head/body type of split.

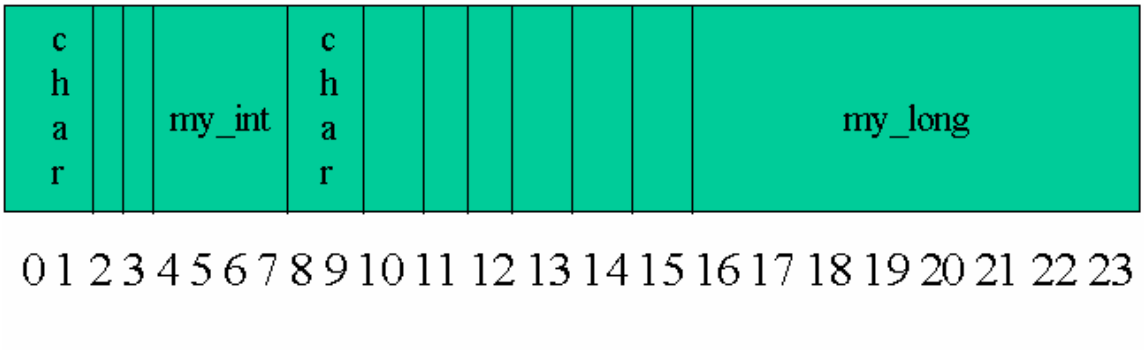
A symbol table is basically a reference table of strings. There is only one copy of each string that exists in the table in the entire database. This is useful for strings which are very common in the database. An example might be the names of the states, or zip codes. One important note about symbol tables is that it can be a concurrency bottleneck if some use cases are attempting to reference data in the table while other use cases are attempting to update the table. This bottleneck can be overcome by pre-creating the symbol table. Populate it with the data before the use cases begin to access it.

Data Member Ordering

Different compilers have different object layout and alignment techniques. Some require that integers be 4 byte aligned, or that longs be 8 byte aligned. Because of these alignment regulations, a persistent object may have padding inserted between data members. This padding is wasted space. Consider the following object on a Tru64 platform:

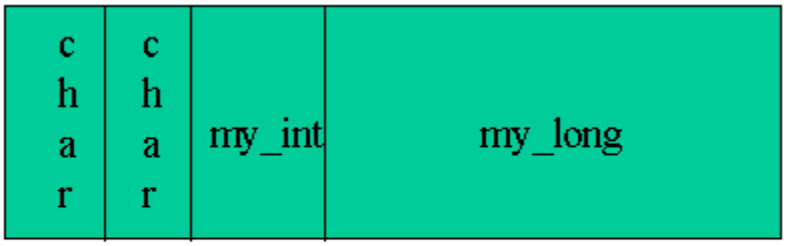
```
class example_part
{
private:
  char[2]     small_string ;
  int         my_int ;
  char[2]     another_string ;
  long        my_long ;
};
```

Here is a pictorial representation of the above:



The above class would be size 24 bytes on a Tru64 platform due to compiler alignment rules. The second data member, "my_int" could not be physically placed immediately after the first data member. Rather, the integer must be 4 byte aligned. Therefore, two bytes of space will exist between the first data member and the second. The long data type must be 8 byte aligned; therefore six bytes of space are wasted padding between the third data member and the fourth. A simple reordering of the class members can result in no space being wasted due to compiler alignment/padding. Here is the class reordered with no wasted space:

```
class example_part
{
 private:
   char[2]      small_string ;
   char[2]      another_string ;
   int          my_int ;
   long         my_long ;
};
```



Again, wasted space due to padding from compiler alignment rules means larger objects. Larger objects mean fewer objects fit per physical page. Fewer objects per page means more page transfers which results in poorer performance. In the example above, we decreased our object from size 24 bytes to 16 bytes. Although 8 bytes may not seem like a lot, if you have thousands of instances of these objects, the overall savings will be great.

Collection Representations

If an application makes use of a large number of collections, then the collections themselves could account for a significant portion of the size of the database and/or the performance of the database application. Therefore, the choice of an appropriate collection representation is of utmost importance. For example, if a collection is being used to hold a class extent and it always has an index, then one should use an index only collection. If there is one primary key that is always queried, a dictionary (rather than a collection) would provide for faster lookups. As mentioned before, if an index exists for a collection, isolation of that index will most likely help to improve performance.

Virtual keyword

If a database contains numerous small objects those objects should avoid having virtual functions if at all possible. The existence of a virtual function causes a class to have a virtual function table (vtbl) pointer. The additional pointer will increase the size of the object. Depending on how small the object was to start with, the addition of the vtbl pointer might add significant overhead to the database. Any derived classes will also incur the overhead of the vtbl pointer. Again, larger objects result in fewer objects per page. Fewer objects per page means more page transfers. More page transfers means more time for an application to execute. Therefore, use virtual functions only when necessary.

Partitioning

Partitioning is a strategy to isolate subsets of objects in different physical storage units. By definition, if two objects are in different partitions, they are de-clustered. The two goals of partitioning are to gain isolation and to increase data density. Isolation is desirable when concurrent access is required. The scope of this paper is not intended to cover concurrency. For that reason our discussion of partitioning will be rather brief.

Although partitioning is intended for isolating objects, its use can improve data density. This may seem, by definition, to be counter intuitive. Let us use an example to illustrate. Imagine a grocery store. If you were in need of a box of cereal, you would go down the cereal aisle. If the grocer has done his job correctly, the aisle (or some number of shelves in the aisle) will be populated ONLY with boxes of cereal. Because other items have been located in their respective aisles/shelves, the entire cereal aisle is dense with cereal. If the grocer had not done the job correctly, a given section of a shelf might have (for instance) boxes of noodles, cans of vegetables, and bags of chips. In this scenario, the shelf does not have good data density for the goal of obtaining a box of cereal. Recall the definition of data density: the proportion of objects within a given storage block that are accessed by a client during some scope. Our scope is to obtain a box of cereal. Our storage block is the aisle or a shelf. If the shelf in question contains many items other than cereal, then we have poor data density. If, on the other hand, we partition the non-cereal items to be in different aisles, then the cereal aisle would contain only cereal and thus a high data density would be achieved.

In the case of ObjectStore, a common area where partitioning can be used is with collections. The collection head can be placed in one physical location; the index in a separate physical location and the items themselves in a third separate location.

Conclusion

The way in which data is loaded into the database can have significant impact on the performance of an application. Careful analysis of the use cases for an application should allow key objects to be identified. Once key objects are identified, a clustering strategy can be planned. Several of the techniques presented here can allow for a clustering strategy that will boost performance far beyond any tuning that might be done after the database is loaded and the application delivered. It is often the case that several techniques can be combined; an application need not restrict itself to the use of just one technique. The goal of clustering is to reduce your working set size; yield higher data density; and reduce the number of pages which need to be transferred between the application and the ObjectStore server.