Semantic Clustering

Karen Shannon
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
shannon@cs.unc.edu

Richard Snodgrass
Department of Computer Science
University of Arizona
Tucson, AZ 85721
rts@cs.arizona.edu

Abstract

Appropriate clustering of objects into pages in secondary memory is crucial to achieving good performance in a persistent object store. We present a new approach, termed *semantic clustering*, that exploits more of a program's data accessing semantics than previous proposals. We insulate the source code from changes in clustering, so that clustering only impacts performance. The linguistic constructs used to specify semantic clustering are illustrated with an example of two tools with quite different access patterns. Experimentation with this example indicates that, for the tools, object sizes, and hardware configuration considered here, performing any clustering at all yields an order of magnitude improvement in overall tool execution time over pure page faulting, and that semantic clustering is faster than other forms of clustering by 20%–35%, and within 25% of the (unattainable) optimal clustering.

The most salient aspect of a tightly coupled persistent object store is that it blurs the distinction between data stored in main memory and data resident on secondary storage. Objects are accessed in a program using such an object store with little or no regard to where the object actually resides [Balch et al. 1989]. If in fact the object has not been cached in main memory, the first access to the object results in an *object fault*, in which the object is read in from disk and made available for access. Generally, objects are *clustered* on disk into segments, and an object fault transfers an entire segment from disk to main memory. We don't consider here objects whose size is greater than the smallest segment, in part because such objects won't benefit from any clustering scheme.

In this paper we present a new approach to clustering that exploits more of a program's data accessing semantics than previous proposals. This approach retains the user's lack of concern for whether an object is cached in main memory, while significantly increasing the performance of the program by simultaneously reducing CPU overhead and disk I/O time.

The next section introduces the tradeoffs inherent in clustering and summarizes previous approaches. We present an overview of our approach, termed *semantic clustering*, in Section 2, with a detailed example appearing in Section 3. Section 4 presents the results of experiments that indicate several performance advantages to semantic clustering. The last section briefly examines how we plan to put this approach into practice in a fairly large programming environment.

1 Implementing Object Faulting

The data model supported by a persistent object store is a (potentially very large) collection of objects, each containing uninterpreted data along with references to other objects. Programs start with a designated root object, traverse some of the embedded references, and make changes to some of the objects encountered. When the program commits, all changes become visible to other programs that use the object store. The runtime system is responsible for moving objects between main memory and secondary storage, and for converting between alternative representations. To the program, all objects are equally accessible; it is the runtime library's responsibility to maintain this fiction in the presence of disparate main memory and disk access speeds.

There are three policies the runtime system must implement. First, how should objects be grouped into segments? Second, when should each object or segment be transferred to or from disk? And third, when should the representation of each object be converted from external form to internal form, and vice versa?

The most obvious implementation simply brings in only the relevant object when it is first accessed by the program. With this approach, termed *strict object faulting*, objects reside in main memory only if they are actually needed, minimizing main memory usage [Straw et al. 1989]. Another advantage is that locking, which is the usual method of concurrency control, can be easily applied to individual objects. Strict object faulting has several disadvantages. Each access to an object must check for *cache residency*. Such checks add significant overhead. Weddell has shown that in a main memory object-oriented DBMS which closely approximates a tightly-coupled persistent object store, a cache residency check on each access can slow computation by as much as 20% [Weddell 1989]. Alternatively, cache residency checking may be done in hardware, say via a virtual memory mechanism. However, such a mechanism may significantly increase the cost of an object fault, since two context switches are generally involved. In either case, each object fault stops the program for disk I/O, increasing turnaround time. Each object fault results in approximately one disk seek, so the number of disk seeks approaches the number of objects cached in main memory. Memory and disk allocation are complex, due to variable object size. Finally, a cached object table is needed to record which objects are cached in main memory and where such objects are located.

At the other end of the spectrum is a much simpler approach, termed *big inhale* [Conradi & Wanvik 1985], in which all possibly accessible objects are read at program initialization. The advantages and disadvantages trade places relative to strict object faulting: once the objects have been read in, the program runs at full speed, with no cache residency checks or disk reads; a cached object table is not needed; and disk I/O is quite efficient, with few seeks. The disadvantages are all performance related: excessive main memory usage, excessive I/O, and excessive locking.

Clustering has been proposed as a compromise. With clustering, the unit of transfer is a *segment*, containing multiple objects. The advantages are that I/O is somewhat more efficient, since segments are generally a multiple of a fixed page size, and memory and disk allocation (and perhaps locking) are simplified. These advantages depend on an assumed correlation of *temporal locality* and *spatial locality*: objects referenced together in time reside together in the same segment. In any case, there are still significant disadvantages: each segment fault stops the program and causes a disk seek, a cached object table is still generally needed, and cache residency checks are still required.

The fundamental issue concerning clustering is assigning objects to segments to obtain a high temporal/spatial correlation. Various approaches have been proposed. One is to cluster on object creation, either by the program specifying another object with which the newly created object is to share a segment (OB2, ObjectStore), or by defining logical clusters and manually inserting newly created objects into them (OB2, ObjectStore, ObServer [Hornick & Zdonik 1987], Versant). The primary disadvantage here is that the clustering decision is localized to the program which creates the object, and is thus hard to implement or change. Another approach is to cluster by object type (PS-Algol [Cockshott et al. 1984], ObServer), with the disadvantage that such clustering generally results in a lower temporal/spatial correlation. Clustering by total closure, in which all objects reachable from some designated object are placed in the same segment (OB2), generally results in excessively large segments. Finally, clustering by segmenting an embedded tree (termed *syntactic clustering*) results in poor performance for programs that don't walk the embedding tree.

All of the proposed clustering schemes still suffer from several drawbacks. First, entire objects are always transferred, even when only a few attributes are needed by the program. Secondly, all accesses must perform cache residency checks, though syntactic clustering obviates the need for such checks across attributes defining the embedded (abstract syntax) tree. Third, clustering is system-wide, and cannot be configured on a program by program basis. Finally, in most realizations of these clustering schemes, the program source code must be changed if the clustering is modified.

We have developed a new approach to clustering that addresses all of these drawbacks, allowing more of the advantages of both the big inhale and strict object faulting to be simultaneously realized. The basic idea behind our approach is to utilize the semantics of persistent data access by individual programs in determining clustering.

2 Semantic Clustering

In this section we introduce a new approach to clustering; the next section will apply it to an example. Many of our design decisions will become clearer when this example is presented.

Semantic clustering is specified manually by the programmer, who is in the best position to know how the program(s) access data. Of course, the programmer can use statistics on attribute access in clustering the data. Clustering is by *partial closure*: a subset of attributes are specified, and all objects reachable from a designated object, termed the *root* of the cluster, via those attributes are placed in the same *cluster instance*. Particular attributes are designated *cut attributes*; the object referenced by such an attribute becomes the root of a separate cluster instance. The attributes are thereby partitioned into *internal* attributes, whose referent object is guaranteed to be coresident in the same cluster as the object containing the attribute, and *external* attributes, for which no such guarantee is possible. This clustering defines an integrity constraint: no two internal attributes of objects in different cluster instances can reference the same object. It has been our experience that defining appropriate internal attributes is straightforward; often the internal attributes form an embedded tree. Also, it turns out that checking this integrity constraint is almost free during the write operation. Note that clustering determines in which cluster instance each object resides. Interestingly, the concepts of cut attributes, internal attributes, and partial closure for clustering also appear in a different guise, and for a different rationale (logical organization verses performance), in the WorldBase system [Wile & Allard 1989, Widjojo et al. 1990].

Clusters are then *fragmented* by specifying a sequence of sets of attributes. All attributes in the first set are placed in the first fragment type. Those remaining attributes that are in the second set are placed in the second fragment type. This continues until all the attributes in all of the object types have been placed into a fragment type. Hence, an object is represented by a collection of one or more *object fragments*, each in a different fragment instance, termed a *segment*. The first object fragment is termed the *identity fragment*; all references from other objects point to this fragment. The identity fragment points to all other fragments of that object, and they all contain pointers back to the identity fragment. In this way, a cluster instance is composed of potentially many segments, each containing some of the attributes of each object in the cluster instance.

Clustering is specified both for the main memory of each program accessing persistent data and also for the data stored on disk. Clustering defines the fragment types to be present in each location, with the runtime system effecting a translation of the data from one clustering arrangement to another as a result of a segment read operation (translating from disk clustering to main memory clustering on input) or an explicit commit operation by the program (translating back to disk clustering on output). The main memory and disk clusterings are designed together to ensure high performance. Generally, a program's main memory is clustered similarly to the disk data, so that minimal translation is needed.

Each program is associated with one or more input and output *ports*, each of which is itself associated with a set of relevant attributes. Programs access stored data through ports, which appear to the program as routines that may be invoked with object identifiers. Invoking an input port causes all objects reachable from the provided object via the port attributes to be logically read into main memory; in practice the objects are faulted in as needed as the program executes. Invoking an output port causes the port attributes of all objects reachable from the provided object via the port attributes to be updated in the store.

When a port routine is invoked, only the relevant segment(s) containing the root object are read in. Access through internal attributes is always permitted. Access through an external attribute requires a cache residency check, which determines whether the object identifier is a *non-persistent reference* (NPR), i.e., a pointer (which must be an even value), or a *persistent identifier* (PID, which we represent as an odd integer). If the latter is the case, an *object fault* occurs, and the runtime system determines whether the segment containing the object is resident in main memory. If not, a *segment fault* occurs, and the cluster instance containing the object is retrieved from disk. In particular, only the relevant segments are retrieved, i.e., those containing attributes of interest to the program. Some segments may not yet exist, if their attributes have not yet been computed, in which case new segments are initialized so that all of the relevant attributes are available to the program. The PID is *swizzled* into an NPR (actually, all internal attributes in the just input segment(s) are swizzled, termed *eager swizzling* [Moss 1990]) and control returns to the program.

At the same time that the indicated cluster instance is retrieved during a segment fault, additional cluster instances may also be retrieved, as specified when the program was clustered. The advantages are two-fold. First, objects in these other segments presumably will be needed later by the program, so they might as well be read in now, potentially saving disk seeks. Second, pre-paging other segments allows more attributes in the program to be designated as internal, reducing (sometimes dramatically) the number of cache residency checks and object faults performed at runtime.

During the write operation though an output port, the port attributes determine which objects, and thus which cluster instances, are written. Additionally, the programmer can assert that certain clusters or fragments are not modified by the program; the output port need not update these segments, saving disk write operations. In order to further reduce I/O time, the programmer can also specify the order the segments are laid out on disk, potentially decreasing the number of disk seeks.

Sets of attributes are used extensively to specify clusters, fragments, ports, pre-paging, and read-only portions. We utilize *structures*, which are collections of object declarations, as a linguistic device to aid in the expression of clustering. Structures can be derived from other structures in various ways, permitting sets of attributes to be expressed indirectly in terms more natural to the user. One additional benefit of these specifications is that they are language-independent, so that multiple programming languages can be made persistent, and can even share data.

The analysis of structure specifications and of clustering specifications, the designation of internal and external attributes, and the generation of code and tables for attribute access, attribute modification, object creation, object faulting, and object output during commit is performed by a data structure compiler. The data structure compiler is responsible for the exact layout of data both on disk and in main memory.

Finally, it is important that clustering not change the semantics of programs; clustering should only impact performance. Since all aspects of data access, modification, and I/O must be moderated, application programs are processed by a preprocessor that inserts the appropriate code to effect the specified clustering. Clustering is easily altered as new insight is gained into how the programs access data, *without* necessitating any changes to the source code of the application program. This property permits graceful evolution of closely cooperating persistent programs [Snodgrass & Shannon 1990].

3 An Example

To illustrate the effectiveness of semantic clustering, we examine an application appropriate for persistent data stores, namely a programming environment. This highly simplistic environment for a functional language contains two tools, among others: a semantic analyzer and a cross referencer, communicating through a central data store, termed a *repository.* We focus on the semantic analyzer, because its algorithms (name resolution being top-down and type resolution being bottom-up, both computing new attributes) and accessing strategies (both traverse the abstract syntax tree) are fairly typical of analysis tools found in a software development environment. We focus also on the cross referencer because its accessing strategy is *not* oriented around the abstract syntax tree, yet is also fairly typical.

We first specify the input and output data structures. We give the actual specifications in their entirety. The details of the specifications are less important than their general flavor. The tool code is given elsewhere [Snodgrass 1989]. Instances of the `abstract_syntax` structure, shown in Figure 1, are created by a parser tool (not described here) and read by the semantic analyzer. The root object, of type `functions`, contains an attribute `syn_funcs`, which is a sequence of `function` objects, each with a name and a body. The function return type is a *class* containing two members, the `int` and `real` types. Similarly, `expression` is a class containing the object `parameterRef` and the classes `constant` and `operation`. The `Enumerated` representation uses integers for the unattributed types `int` and `real`.

Instances of this structure are processed by the semantic analysis tool, which creates instances of the `attributed_syntax` structure, also shown in Figure 1. This structure is derived from the previous structure, and in this case adds three attributes. Name resolution records with each parameter reference the defining occurrence of the parameter, which will have the same name. It also adds each parameter reference to a sequence attached to the parameter; this sequence will be useful during cross reference generation. The `Threaded` representation is a linked list, with each `parameterRef` containing an internally generated attribute that points to the next `parameterRef` in the sequence. Type resolution involves determining the type of each expression. Since `expression` is a class, each member will inherit this attribute. A partial instance of `attributed_syntax` is illustrated in Figure 2.

Finally, we declare two tools and a central repository, and specify their interactions. The input ports are designated with `Pre` and the output ports with `Post`. The output of `CrossRef` is not specified; presumably it is textual output to the screen, to a file, or to a printer. Tools and repositories both have ports. These ports can be connected, with the result that data in the form of structure instances flow out of a tool or repository via an output port, through a connection, and into another tool or repository via an input port. Only two tools are shown; a fully realized programming environment could contain many such tools,

```
Structure abstract_syntax Root functions Is
    functions => syn_funcs: Seq Of function;

    function => syn_name: identifier,
                syn_header: functionDef;
    identifier => lex_token: String,
                  lex_pos: Integer;

    functionDef => syn_ret_type: types,
                   syn_parameters: Seq Of parameter,
                   syn_definition: expression;
    parameter => syn_name: identifier,
                 syn_param_type: types;

    types ::= int | real;
              int =>;  real =>;
    For types Use Representation Enumerated;

    expression ::= constant | parameterRef | operation;
    constant ::= integer_constant | real_constant;
    integer_constant => lex_value: Integer;
    real_constant => lex_value: Rational;
    parameterRef => syn_name: identifier;

    operation ::= binary_operation | unary_operation;
    binary_operation => syn_op: binaryoperator,
                        syn_left: expression,
                        syn_right: expression;
    binaryoperator ::= plus | minus | times | divide;
            plus =>; minus =>; times =>; divide =>;
    unary_operation => syn_op: unaryoperator,
                       syn_argument: expression;
    unaryoperator ::= unaryplus | unaryminus;
            unaryplus =>; unaryminus =>;
End

Structure attributed_syntax Derives abstract_syntax Is
    -- name resolution
        parameterRef => sem_entity: parameter;
        parameter => sem_cross_uses: Seq Of parameterRef;
        For parameter.sem_cross_uses Use Representation Threaded;

    -- type checking
        expression => sem_type: types;
End
```

Figure 1: The `abstract_syntax` and `attributed_syntax` Structures

```
function A(p:int; q:real):real = p + q;
```

Figure 2: An Instance of `attributed_syntax`

connected either to each other or to one or more repositories.

```
Tool SemanticAnalyzer Is
    Pre insyn: abstract_syntax;
    Post outsem: attributed_syntax;
End

Tool CrossRef Is
    Pre insem: attributed_syntax;
End

Repository Rep Is
    Pre insyn: abstract_syntax;        -- computed by parser
    Post outsyn: abstract_syntax;
    Pre insem: attributed_syntax;
    Post outsem: attributed_syntax;    -- sent to optimizer
End

Connect Rep.outsyn To SemanticAnalyzer.insyn;
Connect SemanticAnalyzer.outsem To Rep.insem;
Connect Rep.outsem To CrossRef.inSem;
```

The unclustered environment executes correctly, but is not particularly efficient. We specify additional statements to cluster the data to increase efficiency. While these changes do not impact the tools' source code, they do change the underlying data organization, yielding very efficient object access and modification. We exploit our knowledge of how the tools access and modify data stored in the repository. The semantic analyzer reads in the `functions` object, searches for a `function` of a given name, then makes two passes over the body of the function: a top-down pass to resolve parameter names and a bottom-up pass to type each expression. To avoid reading the bodies of all the functions, we specify that the `syn_header` attribute of `function` is a cut attribute, implying that the `functionDef` object will be the root object of a cluster. The cross referencer tool similarly searches for a function of a give name, iterates through the sequence of parameters searching through one of a given name, then iterates through the sem_cross_uses sequence, touching all references to this parameter. To make this operation efficient, we gather parameterRefs together in a cluster to separate them from the rest of the function body.

To specify this clustering, we first define a *view*, or subset, of the attributed_syntax structure containing the attributes useful to the cross referencer. The most important omission is the syn_body attribute of functionDef.

```
Structure TopSem Views attributed_syntax Is
        functions => syn_funcs: Seq Of function;
        function => syn_name: identifier,
                    syn_header: functionDef;
        identifier => lex_token: String,
                      lex_pos: source_position;
        functionDef => syn_ret_type: types,
                       syn_parameters: Seq Of parameter;
        parameter => syn_name: identifier,
                     syn_param_type: types,
                     sem_cross_uses: Seq Of parameterRef;
        types ::= int | real;
                int =>;  real =>;
        parameterRef => syn_name: identifier,
                        sem_entity: parameter,
                        sem_type: types;
End
```

We group all objects reachable from the functionDef object in a cluster, which we partition into two subclusters, one containing those objects needed by the cross referencer (as specified in the TopSem structure), and one containing the remaining objects. We then fragment these latter two clusters into the syntactic and semantic attributes, the former computed by the parser and read by the semantic analyzer; the latter computed and written by the semantic analyzer. The cross referencer tool will read a subset of these segments. We refine our repository to effect the desired clustering.

```
Repository Rep Refines Rep Is
    Cluster Invariant Is
        Cluster A At function.syn_header Via attributed_syntax Is
            Cluster B At functionDef.syn_parameters Via TopSem Is
                Fragment Between abstract_syntax And attributed_syntax
            End;
            Cluster C At functionDef.syn_definition Via attributed_syntax Is
                Fragment Between abstract_syntax And attributed_syntax
            End
        End
    End;
    -- Order the segments on disk
    Order A, B.1, C.1, B.2, C.2;
End
```

The At construct specifies the cut attribute, the Via clause specifies the attributes that determine object membership in the cluster, and syntactic nesting specifies cluster containment. The Order clause will reduce the number of required disk seeks. Figure 3 illustrates the segments that comprise the stored data for the example function shown in Figure 2. We label each object with a PID; the functions object has a PID of 1. Sets of objects (denoted with "{ }") and sequences of objects (denoted with "< >") are themselves objects. Objects can appear in several segments if they are fragmented (an example is the parameter object with PID 4, whose attributes are located in segments 3 and 4). Object references are eventually represented in main memory as pointers; in the figure we show pointers to objects in other segments with an oval containing the segment number and PID within that segment (e.g., reference 2.8 in the top function object points to the functionDef object in segment 2). The first segment of a cluster is always the identity fragment; it points to the other segments for that cluster (via generated `child` attributes), and the other segments point back to it (via generated `parent` attributes). The parser creates segments 1, 2, 3, and 5; segments 4 and 6 are computed by the semantic analyzer.

We refine the `SemanticAnalyzer` tool, specifying a clustering identical to that of the repository, in order to minimize data format conversions during reading and writing.

```
Tool SemanticAnalyzer Refines SemanticAnalyzer Is
    Cluster Invariant Is
        Cluster A At function.syn_header Via attributed_syntax Is
            Cluster B At functionDef.syn_parameters Via TopSem Is
                Fragment Between abstract_syntax And attributed_syntax
            End;
            Cluster C At functionDef.syn_definition Via attributed_syntax Is
                Fragment Between abstract_syntax And attributed_syntax
            End
        End
    End;
    -- Read and write Clusters B and C at same time as Cluster A
    Use Cluster A Segments;
    -- Specify which portion of output port structure won't change
    For outsem Use No Write Of abstract_syntax;
End
```

We specify that the required segments reachable from cluster `A` should be transferred during I/O. We need these segments anyway, and this clause will reduce disk seeks, residency checks, and object and segment faults. Finally, we assert that the syntactic attributes will not be modified by the semantic analyzer. Conceivably the preprocessor could figure this out, but it affords more type checking to have the programmer explicitly state this assertion. Note that the length of the cluster specification is roughly independent of the length or complexity of the underlying structures. For example, the clustering for the invariant of an Ada semantic analyzer would be very similar to that above.

We cluster the cross referencer tool in a similar manner.

```
Tool CrossRef Refines CrossRef Invariant TopSem Is
    Cluster Invariant Is
        Cluster A At function.syn_header Via TopSem Is
            Cluster B At functionDef.syn_parameters Via TopSem Is
                Fragment Between abstract_syntax And attributed_syntax
            End
            -- No Cluster C needed
        End
    End;
    -- Read Cluster B at the same time as Cluster A
    Use Cluster A Segments;
End
```

This clustering differs from that of the semantic analyzer in three ways. First, this tool does not need all of the attributes in `attributed_syntax`; it only requires those in `TopSem`. Hence, we specify the global data structure within the tool (termed the *invariant*) to be `TopSem`. Second, no cluster `C` is specified, as the attributes in that cluster are not of interest to this tool. Third, since there is no output port, we need not specify the read-only portion.

4 Performance

To determine how effective semantic clustering is in relation to other proposed clustering methods, we ran experiments on six configurations, the preliminary results of which are described here. The source code

```
function A(p:int; q:real):real = p * q;
```

Figure 3: The Layout of the `attributed_syntax` Instance on Disk

remained the same in all configurations; only the policy decisions about how to group objects, when to read or write objects, and when to swizzle were varied. One obvious configuration, strict object faulting, was not included because others have found its performance to be substantially worse than page faulting in the presence of adequate main memory [DeWitt et al. 1990, Stamos 1984].

Configuration 1, termed *pure page faulting*, does not employ application dependent clustering. Instead, objects are grouped on pages based solely on their type. We make the optimistic assumption that the objects of a particular type for each function were grouped together on pages. This assumption, not generally borne out in practice, will minimize the I/O traffic. It also provides a lower bound on the performance of page-based static grouping strategies [Stamos 1984]. In pure page faulting, each access through an attribute that may reference an object is subject to a residency check. If the reference is a PID (this involves simply checking the low-order bit), an object fault occurs, and the PID is looked up in a resident hash table. If the PID is not found there, a page fault occurs, a single page is read in, and all of the objects on the page are placed in the resident hash table. The original attribute is then swizzled. Configuration 2 uses *manual clustering*, where the objects are clustered by executable statements in the parser, which creates the objects. This code arranges that all objects reachable from each `functionDef` object are clustered together in the same segment, which is transferred in whole. Residency checks, object faults, and lazy swizzling (swizzling on access [Moss 1990]) still occur, since the runtime system doesn't know how objects are clustered. However, I/O time is reduced considerably, compared with pure page faulting, since most I/O here is sequential, at least for reasonably sized functions.

In the remaining configurations, the programmer specifies the clustering declaratively in various ways, and the runtime system utilizes this information to increase performance. In the third configuration, syntactic clustering, residency checks and object faults are needed only at the boundaries between segments. As a result, very few residency checks or object faults occur. Configuration 4 uses semantic clustering without fragmentation, here termed *basic semantic clustering.* Compared with syntactic clustering, the partitioning is more sophisticated, as it does not rely on the embedded syntax tree for cluster definition. Configuration 5 uses semantic clustering *with* fragmentation. The clustering statements given in the previous section generate configuration 5; if the `Fragment` clauses are omitted, then configuration 4 results. Configuration 6 assumes an *optimal* clustering strategy that brings in during initialization only those attributes and objects that will accessed. No residency checks are needed; no object faults occur, and no swizzling on input or deswizzling on output is required. Such a strategy is clearly impossible to implement, but provides a convenient lower bound for comparison. Only one seek for read and one for write is needed; the rest of the I/O is sequential.

We instrumented both the semantic analyzer and the cross referencer. We assumed that the repository holds many functions. The tools were executed over five randomly generated function instances stored in the respository; these functions contained from 10 to 100,000 objects. All identifiers in these instances were single characters. The primary metric was total time in seconds to execute the tool. This time may be broken down into the following components: the algorithm, residency checks, processing object faults, reading and writing pages, and swizzling attributes on input and deswizzling attributes on output. Our initial investigations indicated that attribute swizzling/deswizzling in no case totaled more than 2% of the aggregate run time. Hence, we focused on the other components. A somewhat less critical metric is the amount of disk and main memory space required by each configuration.

We ran our tests on an otherwise unloaded NeXT workstation with a 25MHz MC68030 cpu and 16MB of main memory running Version 1.0 of the Mach operating system. All experiments used a local Maxtor 448 MB (formatted) disk, which has an average seek time of 16.5 msec and a raw transfer rate of 4.8 MB/sec, and buffered file I/O. We ensured a cold cache, and flushed the cache to ensure that all modifications were written back to the repository. The source code is in C, and was compiled with GCC. All experiments were run a sufficient number of times to achieve a 95% confidence level of plus or minus 7%.

The algorithm time is not dependent on configuration. To measure it, we loaded the entire database in main memory, swizzled all of the object references, then ran the algorithm multiple times. To measure residency check time, we first measured the number of residency checks performed by each tool in each configuration for each input size, then measured the cost of a single residency check (2.7 μsec). We computed object fault time in a similar manner (a single object fault, which requires a lookup in the object hash table, takes 36 μsec).

I/O time depends on two aspects: the number of seeks and the number of pages read or written. The latter depends on the number of objects transferred, the sizes of the objects, and the page size. We used

Figure 4: Total Execution Time for the Various Configurations

a page size of 1K bytes; segments consist of an integral number of pages. We measured sequential read and write times (2.2 msec and 2.7 msec, respectively) and random read and write times (38 msec and 37 msec), and determined the number of each required by each configuration to arrive at the total I/O time.

Figure 4 shows how the total time varies over function instance size. All runtimes are in seconds. Both axes are logarithmic. The relative difference between approaches is fairly constant across a wide range of number of objects. Below about 1,000 objects, the minimum number of seeks dominates. The optimal case is shown with a darker line.

The following table gives the breakdown for 100,000 objects for the semantic analyzer.

	Page Faulting	Manual Clustering	Syntactic Clustering	Basic Semantic	Semantic w/Fragmentation	Optimal
Algorithm time	6.89	6.89	6.89	6.89	6.89	6.89
# Residency checks	426,977	426,977	2	2	2	0
Residency check time	1.15	1.15	0	0	0	0
# Object faults	100,000	100,000	2	2	2	0
Object fault time	3.59	3.59	0	0	0	0
# Input disk seeks	1,684	2	2	2	2	1
# Input pages	1,684	1,664	1,664	1,665	1,502	873
Input time	63.3	3.6	3.6	3.6	3.3	1.9
# Output disk seeks	1,684	1	1	1	2	1
# Output pages	1,684	1,663	1,663	1,663	791	477
Output time	61.7	4.5	4.5	4.5	2.2	1.3
Total time	137	19.9	15.2	15.2	12.5	10.3

Pure page faulting is by far the slowest, due primarily to the number of disk seeks performed. Each object fault to a non-resident page causes a random page to be read, preceded by a disk seek. Recall that this number is a lower bound, since we assumed each page contained only objects relevant to the algorithm, which the runtime system had no way of ensuring. Manual clustering effects almost an order of magnitude improvement, as the number of seeks decreased to 3, independent of the size of the input. However, the time to perform residency checks and objects faults becomes more important; this time approaches that of the algorithm, and accounts for almost one-fifth of the total time. Syntactic clustering informs the runtime system of the clustering, reducing residency checks and object faults from an amount linear in the number of objects to 2, independent of the size of the input. Basic semantic clustering in this case takes exactly as long as syntactic clustering, because the accessing patterns of the two configurations are identical. Fragmentation cuts the time down by another 18%, because the semantic attributes are not read and the syntactic attributes are not written. The optimal (and unattainable) clustering is faster still, by 18%, in part because fewer seeks are required (one input and one output, verses two of each). More importantly, fewer attributes, and hence fewer pages, need to be transferred: the values of constants and the operator attribute of operation objects are not needed, and so are not transferred in this configuration, and various internal pointers such as those employed between fragments also are not needed.

The following table gives the breakdown for the cross referencer tool, again for 100,000 objects.

	Page Faulting	Manual Clustering	Syntactic Clustering	Basic Semantic	Semantic w/Fragmentation	Optimal
Algorithm time	0.74	0.74	0.74	0.74	0.74	0.74
# Residency checks	4,640	4,640	2	2	2	0
Residency check time	0.01	0.01	0	0	0	0
# Object faults	4,638	4,638	2	2	2	0
Object fault time	0.17	0.17	0	0	0	0
# Input disk seeks	665	2	2	2	3	1
# Input pages	665	1,664	1,664	656	820	491
Input time	25.0	3.65	3.65	1.48	1.87	1.09
Total time	25.9	4.57	4.39	2.22	2.61	1.83

Cross referencing takes an order of magnitude less time (in large part due to the `sem_cross_uses` sequence so conveniently created during name resolution). Interestingly, pure page faulting has an advantage when used with the cross referencer tool, because the tool touches objects of only a few object types. Hence, the data is accidentally clustered. However, seek time still dominates. Compared with syntactic clustering, basic semantic clustering is significantly faster, because it avoids reading the body of the function. Fragmentation increases the size of objects slightly to accommodate pointers connecting objects in different segments, resulting in slightly lower performance (the increase is more the compensated by the decrease observed in semantic analysis).

The differences in the space requirements aren't as dramatic. The following table how much disk space in MB is required to store a 100,000 object function instance, and how much main memory is used by each of the tools for each configuration. Basic semantic clustering reduces the main memory usage as compared with the other clustering approaches, and semantic clustering with fragmentation generally trades disk space for execution time. The disk space overhead imposed by fragmentation is accentuated by the small objects used in this experiment. If somewhat larger objects were used, then the relative overhead of the additional pointers required by fragmentation would be less.

	Page Faulting	Manual Clustering	Syntactic Clustering	Basic Semantic	Semantic w/Fragmentation	Optimal
Disk space	1.68	1.66	1.66	1.67	2.29	1.65
Sem. analy. main mem.	1.68	1.66	1.66	1.67	2.29	1.35
Cross ref main mem.	0.67	1.66	1.66	0.66	0.82	0.49

This experiment indicates that the following factors may be ranked as follows according to their contribution to the total execution time of each tool: disk seeks (35 msec), pages transferred (2.2–2.7 msec), object faults (36 μsec), and residency checks (2.7 μsec). We now discuss strategies to minimize each in turn.

Minimizing disk seeks is difficult when multiple tools exhibit different accessing patterns. Pure page faulting suffered because page faults were to random pages, requiring a seek for each page read (a total of 3368 seeks in semantic analysis and 665 in cross referencing). Increasing the page size would substantially improve the performance of this configuration for large functions. A page size of 32K bytes would for example cut the semantic analysis execution time to approximately one-fourth and the cross referencer time to one half for functions of 10,000 objects, at the cost of significant wasted main memory (416KB of main memory needed to store 171 KB of data). However, pure page faulting even with this change would not be competitive with the other configurations. Larger page sizes would probably not substantially help the other configurations. Preliminary studies by others have yielded the same conclusion [DeWitt et al. 1990].

The other five configurations required from 1 to 4 seeks for both tools, independent of the number of objects transferred. Semantic clustering minimizes disk seeks in three ways: segments are defined in such a way as to select exactly those objects and attributes of interest to each tool, entire segments are transferred at a time, and the segments are manually ordered on disk to allow several segments to be read in one sequential pass. Even though fragmentation increases the number of segments required (manual, syntactic, and pure semantic clustering required one segment per function, while fragmentation required four), the number of disk seeks required by fragmentation were comparable, and in some cases identical, to these other configurations.

Minimizing pages transferred involves not transferring unneeded objects and attributes. Syntactic clustering suffered in the semantic analyzer because it had to read the uninitialized semantic attributes; it suffered in the cross referencer because it had to read the entire function body, much of which was never used. Fragmentation is the key to minimizing the number of pages read and written, effecting a time savings of 20% for the semantic analysis tool.

Minimizing object faults and residency checks requires knowledge by the runtime system of what objects will be co-resident. In pure page faulting the time to perform these two bookkeeping tasks approached that required to execute the semantic analysis algorithm (disk I/O time still overwhelmed these other components). In manual clustering these bookkeeping tasks accounted for a quarter of the total time. The other configurations all performed exactly two residency checks and two object faults, independent of the number of objects. This low count is due in part to the simplicity and small number of the tools; we envision that other tools may require somewhat more checks. Semantic clustering offers flexibility in clustering objects, and also allows enclosing clusters to be simultaneously read or written by a tool, permitting more attributes to be internal and thus not subject to residency checks or object faults.

As processing speed continues to improve at a faster rate than disk access speed, this ranking of factors should continue to hold, and in fact the goal of reducing the number of disk seeks and page transfers will become increasingly important.

5 Summary and Future Work

In this paper we introduced a new approach to clustering in a persistent object store. Semantic clustering differs from other proposed clustering methods in the following ways.

- Clustering is by partial closure, via a subset of the attributes.
- Objects are then fragmented in a controlled fashion, splitting attributes across segments.
- The program's main memory is clustered to eliminate most residency checks and to effect prepaging of objects.
- The ordering of segments on disk and the portion of main memory objects that are modified by a tool can both be specified; this information is used to reduce I/O time.
- Structures are utilized as a declarative linguistic device to aid the specification of clustering.
- We insulate the source code from changes in clustering, with the result that the clustering impacts performance only (the tool developer need not be concerned with clustering).

We presented results of experiments on a semantic analyzer and a cross referencer, which exhibit quite different object access patterns. While semantic clustering exploits knowledge of these access patterns, it should work well with all of the tools that access the repository. The total execution time for both tools is shown below, along with the overhead imposed by making the tool persistent (calculated as total time minus algorithm time).

	Page Faulting	Manual Clustering	Syntactic Clustering	Basic Semantic	Semantic w/Fragmentation	Optimal
Both tools	163	24.5	19.6	17.4	15.1	12.1
Persistent Overhead	155	16.9	12.0	9.8	7.5	4.5

This experiment indicates that, for the tools, object sizes, and hardware configuration considered here, performing any clustering at all yields almost an order of magnitude improvement in overall tool execution time over pure page faulting, and that semantic clustering is faster than other forms of clustering by 20%–35%, and within 25% of the (unattainable) optimal clustering.

The drawbacks of semantic clustering are (a) the clustering must be correctly specified by the programmer (a bad clustering is often worse than none at all), (b) new or modified tools may be inconsistent with the current clustering in place, resulting in either slow tools or repository reorganization (this tradeoff may be managed by effective tools [Snodgrass & Shannon 1990]), (c) fragmentation requires more disk and main memory space to store cross pointers between segments, and (d) the runtime system is somewhat more complicated than pure page faulting, but similar to that needed for syntactic clustering,

We have developed a formal semantics for the clustering constructs, and are now implementing a runtime system that fully supports semantic clustering. We will then apply this clustering to a large programming environment [Snodgrass 1990]. This environment includes a data structure compiler of the form mentioned in Section 2, as well as a window-based debugger, an assertion checker, an unparser, and a cross reference generator. The dozen programs contain some 20 persistent tools, for a total of approximately 100K lines of source code. This code will require few changes to use the new runtime system, and will provide an opportunity for a much more complete evaluation of this approach. We also plan to apply semantic clustering to specific persistent languages such as Modula-3 [Hosking & Moss 1990] as alternatives to the language-independent constructs illustrated in Section 3.

6 Bibliography

[Balch et al. 1989] Balch, P., W. P. Cockshott and P. W. Foulk. *Layered Implementations of persistent object stores. Software Engineering Journal*, Mar. 1989, pp. 123-131.

[Cockshott et al. 1984] Cockshott, W., M. Atkinson, K. Chisholm, P. Bailey and R. Morrison. *Persistent Object Management Systems. Software–Practice and Experience*, 14 (1984), pp. 49-71.

[Conradi & Wanvik 1985] Conradi, R. and D.H. Wanvik. *Mechanisms and Tools for Separate Compilation.* Technical Report 25/85. The University of Trondheim, The Norwegian Institute of Technology. Oct. 1985.

[DeWitt et al. 1990] DeWitt, D.J., P. Futtersack, D. Maier and F. Velez. *A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems*, in *Proceedings of the Conference on Very Large Databases.* 1990.

[Hosking & Moss 1990] Hosking, A.L. and Moss, J.E.B. *Towards Compile-Time Optimizations for Persistence*, in this proceedings.

[Moss 1990] Moss, J.E.B. *Working with Persistent Objects: To Swizzle or Not to Swizzle.* COINS Technical Report 90-38. Department of Computer and Information Science, University of Massachusetts. May 1990.

[Snodgrass 1989] Snodgrass, R. *The Interface Description Language: Definition and Use.* Rockville, MD: Computer Science Press, 1989.

[Snodgrass 1990] Snodgrass, R. *IDL Toolkit Release 4.2.* Department of Computer Science, University of Arizona, Tucson, AZ, 1990.

[Snodgrass & Shannon 1990] Snodgrass, R. and K. Shannon. *Fine Grained Data Management for Evolving Tools in a Software Development Environment*, in *Proceedings of the Symposium on Software Development Environments.* Irvine, CA: Dec. 1990.

[Stamos 1984] Stamos, J.W. *Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. ACM Transactions on Computer Systems*, 2, No. 2, May 1984, pp. 155-180.

[Straw et al. 1989] Straw, A., F. Mellender and S. Riegel. *Object Management in a Persistent Smalltalk System. Software–Practice and Experience*, 19, No. 8, Aug. 1989, pp. 719-737.

[Weddell 1989] Weddell, G.E. *Efficient Property Access in Memory Resident Object Oriented Databases.* Technical Report CS-89-49. University of Waterloo. 1989.

[Wile & Allard 1989] Wile, D.S. and D.G. Allard. *Aggregation, Persistence, and Identity in Worlds*, in *Proceedings of the Workshop on Persistent Object Systems.* Jan. 1989.

[Widjojo et al. 1990] Widjojo, S., R. Hull, and D.S. Wile. *A Specificational Approach to Merging Persistent Object Bases*, in this proceedings.