# Blackboard Retrieval Mechanisms in GBB

Kevin Q. Gallagher Daniel D. Corkill

December 1988
COINS Technical Report 88-39

# Blackboard Retrieval Mechanisms in GBB

**Kevin Q. Gallagher and Daniel D. Corkill**

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

December 1988

**Abstract**

Associative retrieval of blackboard objects is central to the blackboard paradigm. When numerous objects reside on the blackboard, retrieval costs can be reduced signifcantly by limiting search to the blackboard area containing the desired objects. Partioning the blackboard into levels is one means of reducing search. The Generic Blackboard Development System (GBB) further reduces search by representing each level as a highly-structured, n-dimensional volume. Blackboard objects occupy an extent within the n-dimensional volume based on the values of *dimensional index* attributes. In this paper, we detail the blackboard representation and retrieval algorithms used by GBB. These are based on range searching techniques [1]. GBB also facilitiates tuning the performance of blackboard retrieval for a particular application. The representation and retrieval algorithms can be changed without recompiling the application. We also present general guidelines (based upon blackboard density and blackboard insertion/retrieval ratios) for tuning the representation to produce the best performance.

## 1 Introduction

Associative retrieval of blackboard objects is central to the blackboard paradigm. It is used to provide *anonymous communication* among knowledge sources (KSs) by allowing KSs to look for relevant information on the blackboard rather than receiving the information via direct invocation by other KSs. However, the blackboard provides more than this anonymous communication channel among KSs. Objects on the blackboard often have significant *latency* between the time they are placed on the blackboard and the time they are retrieved and used by another KS. If it were not for this latency, the blackboard could be "compiled away" into direct calls among KSs by a configuration-time compiler. This latency in blackboard objects indicates that the blackboard also serves as a *memory* for the KSs. Objects are held on the blackboard to be used when and if they are needed by the KSs.

This research was sponsored in part by donations from Texas Instruments, Inc., by the Office of Naval Research, under a University Research Initiative Grant (Contract N00014-86-K-0764), and by the National Science Foundation under CER Grant DCR-8500332

When numerous objects reside on the blackboard, retrieval costs can be signifcantly reduced by limiting search to the blackboard area containing the desired objects. Partioning the blackboard into levels is one means of reducing search. The Generic Blackboard Development System (GBB) [2,3] further reduces search by representing each level as a highly-structured, n-dimensional volume. Blackboard objects occupy an extent within the n-dimensional volume based on the values of *dimensional index* attributes. In this paper, we detail the blackboard representation and retrieval algorithms used by GBB. These are based on range searching techniques [1]. We also present general guidelines (based upon blackboard density and blackboard insertion/retrieval ratios) for tuning the representation to produce the best performance.

## 2 GBB Blackboard and Unit Representation

GBB represents the blackboard database as a hierarchical tree[1] of nested *blackboards*, the leaves of which are primitive blackboard elements called *spaces*. For example, spaces would be used to implement the problem solving levels in the Hearsay-II speech understanding system [4]. GBB views each space as a structured n-dimensional volume. This space dimensionality provides a *metric* for positioning blackboard objects, called *units*, onto the blackboard in terms that are natural to the application domain. A space dimension can be either *ordered*, real numbers in some interval; or *enumerated*, members of a set of labels.

Units occupy some n-dimensional extent within the space's dimensionality. A unit is located on a space based on the values of its *dimensional indexes* (often abbreviated to just *indexes*). Each index corresponds to a space dimension.[2] Each unit defines how its index values will be computed from its slots (attributes). For example, a single slot may contain both the $x$ and $y$ dimensional indexes for a unit.

There are two types of indexes, *scalar* indexes and *composite* indexes. A scalar index represents a single "atomic" value. For ordered dimensions the scalar index types are *points* (a single number) and *ranges* (two numbers representing minimum and maximum values). For enumerated dimensions an atomic index value is a single label.

A composite index represents a group of atomic index values. Composite indexes are divided into two types, *sets* and *series*. A set index is an unordered group of dimensional index values. A series index is a group of dimensional index values where the order of the elements is determined by the value of another dimensional index. For example, suppose an application is monitoring a chemical reaction. A series of temperature observations over time would be implemented as a series index. The two dimensional indexes in this example are *time* and *temperature*. *Time* is called the *ordering index* — it is the index by which the *temperature* observations are ordered. Several non-ordering indexes may be ordered by the same ordering index (e.g., a series of temperature and pressure observations over time), however the current implementation does not support more than one composite index.

It is important to note that the distinction between scalar and composite values is not a property of space dimensions; it is a property of indexes. That is, each unit class

[1] Actually, it can be a forest.

[2] The set of indexes need not exhaust the set of space dimensions. However, such units can not be retrieved using the missing dimensions.

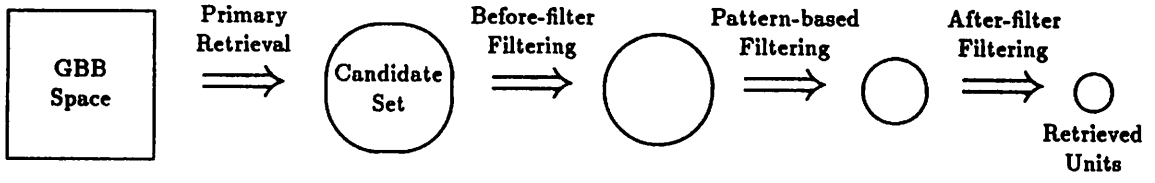



Figure 1: GBB's Unit Retrieval Steps.

specifies whether an index is scalar or composite. To summarize:

- Spaces have dimensions.
- Units have indexes.
- An index is a particular value or set of values for a dimension.

In database terms, indexes are *keys*, but in the case of composite indexes, a single key can be quite complex.

# 3 Overview of Blackboard Retrieval

GBB retrieves units from a space based on a *retrieval pattern*. The pattern specifies an n-dimensional volume of the blackboard in which the desired units will be found. In addition to the pattern, a retrieval request can specify two procedural filters, called the *before-filter* and the *after-filter*. These filters check for conditions that can not be expressed in the pattern. For example, a filter might compute a complex predicate on the unit, or check attributes that are not represented in the dimensionality of the space.

The retrieval process is divided into four steps: *primary retrival*, *before-filter* filtering, *pattern-based* filtering, and *after-filter* filtering (Figure 1). To simplify the exposition, we will describe each step as if the result of one step is passed on to the next step. In fact, the steps are interleaved to avoid creating temporary lists of intermediate results. In the following description, we will concentrate on the retrieval mechnanisms used for ordered dimensions.

The primary retrival step selects a set of units that might satisfy the pattern. This is called the *candidate set*. Each member of the candidate set is then passed through the three filtering steps. Units that satisfy all the remaining steps are collected into a list and returned as the result of the retrieval. Because the filtering steps consider only units in the candidate set, the primary retrieval step must select *all* units that could possibly satisfy the remaining steps.

The pattern-based filtering step compares each unit to the pattern to see if the unit satisfies the constraints of the pattern. This step is necessary because membership in the candidate set does not guareantee that a unit will satisfy the pattern. The before-filter and after-filter filtering steps simply apply the filter predicates to each unit. If a predicate returns false then the unit is removed from further consideration. The goal of efficient retrieval is to efficiently minimize the number of units in the candidate set which do not satisfy the pattern. This will minimize the number of units that must be examined in the remaining steps.

Each space has a *unit mapping* which specifies how the storage of units on that space is implemented. The simplest implementation strategy is to store all a space's units in a list. In this case the primary retrieval step is very simple — the candidate set is simply the entire list of units on the space. However, the filtering steps must be applied to every unit on the space. This is an ineffecient strategy when many units do not satisfy the retrieval constraints.

The problem with the simple list strategy is that the primary retrieval doesn't reduce the candidate set at all. A better strategy is to partition a space's storage into buckets or cells based on its dimensional attributes. (This is similar to the partition and grid range searching strategies [1].) The primary retrieval step can then examine the pattern to determine which buckets should be included in forming the candidate set. These buckets are called *candidate buckets*. The candidate set is the union of the contents of all the candidate buckets.

The bucket technique is appropriate for ordered indexes, which have the property of *neighborhood*. That is, there is a notion of one index value being "near" another index value. Examples of such indexes are physical dimensions such as $x$ and $y$ location and *time*. A hash table can not easily represent this neighborhood relationship among index values.

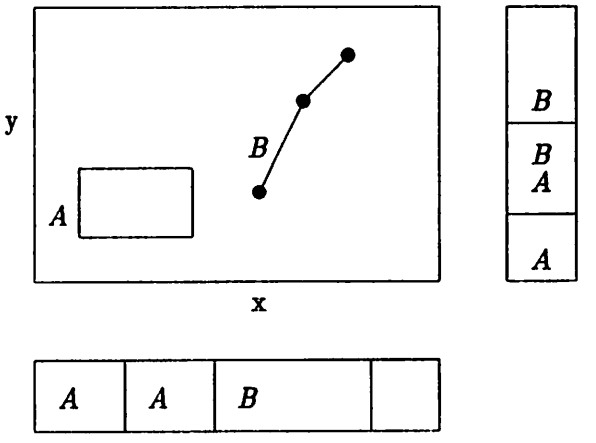
On the other hand, a hash table *is* appropriate for indexes which are unordered such as GBB's enumerated dimension type, which is simply an unordered set of labels.

# 4 Details of Primary Retrieval

As mentioned above, there is a *unit mapping* which specifies exactly how the storage is to be arranged for each unit class that can be stored on space. In particular, the unit mapping specifies what space dimensions will be used in the primary retrieval step. It specifies how to partition these dimensions into separate buckets and the exact arrangement of the data structures.

If a unit mapping specifies that *no* space dimensions are to be used to locate units on a space then the space is implemented as a list. To retrieve units from the space in this case the three filtering steps are applied to every unit on the space because the candidate set is the entire list of units. Units that satisfy the filtering steps are then collected into a list of retrieved units. This representation is desirable only if there are very few units on the space. Typically, however, there will be hundreds or thousands of units on a space.

When one or more space dimensions are specified for the unit mapping then the space is implemented as one or more arrays. Each array dimension represents one space dimension and each array element represents one bucket. Each element contains a list of all the units which fall into that bucket. For example, if a space has the dimensions *time*, $x$, and $y$ then one possible unit mapping would use three vectors, one for each of the space dimensions. Another representation would use a two-dimensional array for $x$ and $y$ and a vector for *time*. Note that a unit may fall into several buckets in a single array depending on the values of the dimensional indexes for that unit, so the lists of units in each bucket are not disjoint. For example, a dimensional index value may be a range which spans a bucket boundary (Figure 2, unit $A$), or the elements of a composite index may fall into different buckets (Figure 2, unit $B$).




A two dimensional space (x, y) containing two units ($A$ and $B$). The boxes on the right and the bottom represent two vectors of buckets and show which buckets the two units fall into. Note that the bucket sizes need not be uniform.

Figure 2: Mapping Units to Buckets

A unit mapping may specify that only a subset of the space dimensions will be used in the primary retrieval step. For example, another unit mapping for the three dimensional space above would be two vectors, one for $x$ and one for $y$.[3]

By examining the pattern, GBB can determine the candidate buckets in each array. This is a straightforward computation based on the values for each dimensional index in the pattern. A minor complication arises when elements of a composite index in the pattern overlap one another. Without special attention, a bucket could be included more than once because separate index elements fell into the same bucket. To avoid unnecessary search, duplicate buckets are removed.

To aid in retrieval processing, Each unit has a special internal slot called the *mark* slot which contains a small integer. During the retrieval process, this slot is used to record the status of each unit.

If a space is implemented as a single array then the candidate set is simply the union of the contents of all the candidate buckets. To avoid creating temporary lists each candidate bucket is processed one at a time. Each unit in the bucket is tested by the three filtering steps and marked as having passed or failed. If a unit has already been marked as tested it is not tested again. Those units which pass the filtering steps are collected into a list of retrieved units.

If a space is implemented as more than one array then the candidate set is the intersection of the contents of the candidate buckets for each array. A straightforward intersection algorithm would require time $O(n^m)$, where $n$ is the number of units on the space and $m$ is the number of separate arrays, as well as space $O(n)$. By using the mark slot associated with each unit we can reduce the time to $O(n)$ and the space to a constant.

[3]One reason this might be done is that a space may store several different unit classes and all space dimensions may not be useful primary retrieval keys for all classes.

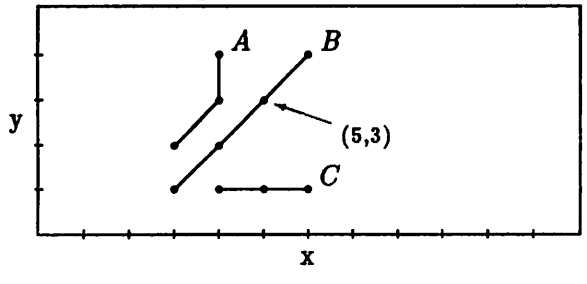
```
;; Clear the mark in the first array's candidate buckets.
(clear-unit-marks (bucket-selection (first space-arrays)))

(let ((count 0))

   ;; Update the mark in the remaining arrays' candidate buckets.
   (dolist (array (rest space-arrays))
      (dolist (unit (bucket-selection array))
         (when (= (get-unit-mark unit) count)
            (incf (get-unit-mark unit))))
      (incf count))

   ;; Collect the units in the intersection.
   (dolist (unit (bucket-selection (first space-arrays)))
      (when (= (get-unit-mark unit) count)
         (filter-and-collect unit))))
```

Figure 3: "Tag-based" Intersection

This "tag-based" intersection is computed as follows. Select one array as the initial array and set the mark to zero for all units in its candidate buckets. Then consider each unit in the second array's candidate buckets. If the unit's mark is zero then set the mark to one otherwise leave it unchanged. Then consider each unit in the third array's candidate buckets. If the unit's mark equals one then set the mark to two. This continues until we have exhausted all the arrays. Finally, return to the initial array and examine each unit in its candidate buckets. If the unit's mark is equal to $n$ (where $n$ is the number of arrays) then the unit is a member of the candidate set and the three filtering steps are applied. (See Figure 3.) Because the mark was initialized to zero for each unit in the initial array's candidate buckets, only those units that fall into the candidate buckets for all arrays will have the proper value in their mark slots.

As in the single array case above, each unit in the bucket is tested by the three filtering steps and marked as having passed or failed. If a unit has already been tested it is not tested again. Those units which pass the filtering steps are collected into a list of retrieved units.

To illustrate the tradeoffs, consider the three tracks ($A$, $B$, and $C$) depicted below, and suppose the application is looking for tracks which pass through the point (5,3).




If the space is represented simply as a list of units then the primary retrieval step retrieves all three units, which must be compared with the pattern. If the space is represented as two vectors (one for $x$ and one for $y$), then the primary retrieval step selects all units that occupy the $x = 5$ bucket (in this case B & C). This set is intersected with the set of units

occupying the $y = 3$ bucket (A & B), for a primary retrieval result set (B). Pattern-based filtering is then applied to each element of this result set.

# 5 Details of Pattern-Based Filtering

The pattern-based filtering step compares, in detail, each unit with the pattern to determine if the unit satisfies the constraints of the pattern. The pattern primarily specifies an n-dimensional blackboard volume (or set/series of volumes) in which the desired units will be found. The constraints imposed by each dimension are tested independently. If a unit fails to satisfy the constraints of any dimension, the unit is eliminated from further consideration. The pattern specifies an *element match* criterion which determines how units are compared to the pattern. The four element match criteria are:

**Exact** The unit must exactly match the pattern.

**Includes** The unit must entirely include the pattern.

**Within** The unit must be entirely within the pattern.

**Overlaps** The unit must overlap with the pattern.

The distinctions between the different match criteria are only meaningful when one or both of the index elements from the pattern and the unit are ranges. If both the pattern's and the unit's index elements are points or labels then these match criteria are all equivalent.

Several options are available to control the comparison of composites. The *match* and *mismatch* options apply to both set and series indexes. The remaining options only apply to series indexes.

**Match** This determines how many index elements in the pattern must be matched by index elements in the unit.

**Mismatch** This is the upper limit on the number of index elements in the pattern that may be mismatched with index elements in the unit.

**Skipped** This is the upper limit on the number of index elements in the pattern that may be skipped in the unit.

**Before and After Extras** These options determine how many index elements may appear in the unit either before or after the composite index in the pattern.[4]

**Contiguous** This determines whether the composite index elements that match must be contiguous or whether mismatching elements may appear between the matching elements.

Note that, the index element to index element comparisons are still governed by the element match criterion specified in the pattern.

As with unit indexes, indexes in the pattern can be of type scalar, set, or series. So, the following six combinations of unit index and pattern index can occur. The order here is not important. (Comparing a scalar unit index with a set pattern index is treated identically to comparing a set unit index with a scalar pattern index.)

[4] For example, suppose the pattern has a series of *temperature* observations starting at *time* = 6 and the unit has a series starting at *time* = 4. This unit would pass only if the the before extras option allowed two or more before extras.

| | |
|---|---|
| Scalar/Scalar | Set/Set |
| Scalar/Set | Set/Series |
| Scalar/Series | Series/Series |

To avoid subtle conceptual errors, Scalar/Series comparisons are not allowed.[5] We consider the comparison rules for the remaining five cases below.

**Scalar/Scalar** In the simplest case, when an index is a scalar in both the pattern and the unit, GBB simply compares the value of the index in the unit with the value of the index in the pattern to see if they match with respect to the match criterion from the pattern.

**Scalar/Set** The comparison of a scalar index with a set index reduces to a set membership test. This is the case when the index is a scalar in the unit and a set in the pattern or vice versa. If the scalar index is a member of the set index then the comparison succeeds. Each index element from the set is compared with the scalar index element in the same way as for scalar/scalar comparison above.

**Set/Set and Set/Series** For both these cases the comparison is treated as a set intersection. The size of the intersection set must be greater than the number of matches required by the pattern. In addition the number of elements from the pattern that are not in the intersection set must be less than or equal to the *mismatch* criterion from the pattern.

**Series/Series** In the series/series case the index elements for *corresponding* values of the composite index are compared. Continuing the *time/temperature/pressure* example from above, the values for *temperature* and *pressure* at *time* = 6 in the pattern would be compared with the corresponding values for *temperature* and *pressure* at *time* = 6 in the unit. This continues for each element in the pattern. The unit will pass the pattern-based filtering step if the match, mismatch, skipped, before extras, and after extras criteria are satisfied.

Several patterns can be conjoined. The effect of this is that a unit must satisfy all the patterns to pass the unit/pattern comparison. For conjoined patterns the primary retrieval is based on all the component patterns. Then, during the pattern-based filtering step, GBB will compare each unit to the simplest component patterns first in the expectation that this will "cheaply" eliminate some units early.

# 6 Comparison with Other Representations

The partition and grid techniques used in GBB work best when the units are located so that the buckets are uniformly loaded. Two other representation strategies that have been proposed for blackboard database systems are the *k*-d tree [1] and the quadtree [5]. They are appealing because they readily adapt to the data, providing more selectivity where

[5]For example, in comparing a scalar pattern index with a series unit index, the composite index of the series may not be present in the scalar. It is not immediately apparent whether this situation should be treated as an error. However, the restriction causes the user to be explicit about what is intended. Transforming one type of index to another is a simple operation, so this is not a great limitation.

there is more data, and give good performance when little is known, *a priori*, about the distribution of data or queries.

Both quadtrees and $k$-d trees are hierarchical data structures which recursively divide space. There are two general types of quadtrees: *region quadtrees* and *point quadtrees*. The region quadtree is inappropriate for a blackboard database because the entire quadtree represents just one object. The point quadtree is a multidimensional generalization of a binary search tree. Each node in the tree stores one datapoint and has $2^n$ children. Each child represents one 'quadrent' of the space represented by its parent.

The $k$-d tree is also a generalization of a binary search tree. At each level of the tree a different dimension is tested when determining which branch to take. For example, in the two dimensional case the $x$ dimension is tested at even levels in the tree and the $y$ dimension is tested at odd levels. The *adaptive* $k$-d tree is a refinement of the $k$-d tree where, instead of cycling through the dimensions in a fixed order, choose the dimension that best divides the datapoints in the best way for each node.

While the quadtree and $k$-d tree may use less space than the grid method, there is no speed advantage. The grid method requires $O(R)$ steps on average (where $R$ is the number of objects found), but quadtrees and $k$-d trees require $O(R + \log N)$. In addition, deletion and balancing are quite complicated and time consuming operations on the quadtree and $k$-d tree.

# 7 Guidelines for Blackboard Representation

As our performance tuning experiments indicate, selecting an appropriate blackboard representation (the unit mapping) can have a significant effect on application performance. Here are some general guidlines or hints on acheiving the best performance in GBB by tuning the blackboard representation.[6] Many of them are simple common sense. As with most efficiency techniques, the law of diminshing returns applies. For example, changing from a simple list implementation to a single vector implementation will improve search speeds by 50% to 90%.

- When a small number of units will be created on a space the simple "list" implementation is best because of its low overhead. The threshold is somewhere between 5–8 units with complicated indexes (i.e., composite indexes with several elements) and 20–25 units with simple scalar indexes.
- When all units tend to span the entire range of a space dimension that dimension should be left out of the unit mapping.
- If one dimension evenly divides the units on a space it should be included in the unit mapping even if it only has a few distinct values.
- If the units have complicated indexes then the increased selectivity of the grid (n-dimensional array) is well worth the additional space required. Searches on a 3-dimensional grid have run up to 40% faster than equivalent searches on three vectors.

[6] These are based on experiments run on a Texas Instruments Explorer II workstation. Additional experiments are required to see if our findings extend to other machines.

- Conversely, if the units have scalar indexes then the partition strategy (n vectors) tends to be almost as good as the grid strategy.
- If the units are distributed on a diagonal with respect to the space dimensions, consider using indexes that are a transformation (e.g., rotation) of the "actual" values. This can be done efficiently using the index structure mechanism in GBB. We are considering adding such transformations as a built-in feature of GBB.

# 8 Summary

Reducing the cost of blackboard retrieval can significantly increase the performance of blackboard-based applications. In the Generic Blackboard Development System (GBB) we have provided application developers the mechanisms for high-performance retrieval operations.

GBB provides several types of blackboard representations: a simple list, a single array of buckets partitioned along one or more dimensions, and intersections of several arrays. Application developers can match the performance characteristics of the different representations to their particular application to acheive the best overall performance. In addition, the blackboard representation can easily be changed to accommodate changes in the application's data characteristics without requiring changes to any application code.

# References


[1] Jon L. Bentley and Jerome H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4):397–413, December 1979.

[2] Kevin Q. Gallagher, Daniel D. Corkill, and Philip M. Johnson. *GBB Reference Manual.* Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, GBB Version 1.2 edition, September 1988. (Published as Technical Report 88-66, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, September 1988.).

[3] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, Pennsylvania, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[4] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[5] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, June 1984.

[6] Daniel D. Corkill, Kevin Q. Gallagher, and Philip M. Johnson. Achieving flexibility, efficiency, and generality in blackboard architectures. In *Proceedings of the National Conference on Artificial Intelligence*, pages 18–23, Seattle, Washington, July 1987.

(Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 451–456, Morgan Kaufmann, 1988.).

[7] Daniel D. Corkill and Kevin Q. Gallagher. Tuning a blackboard-based application: A case study using GBB. In *Proceedings of the National Conference on Artificial Intelligence*, pages 671–676, St. Paul, Minnesota, August 1988.