MARIAN Searchers

Definition: A searcher is a class manager that can map abstract descriptions into weighted object sets of matching class instances.

Types (in current use)

- Class-based
  - Superclass of disjoint subclasses
- Link-based
  - Unweighted (flat) link
  - Weighted link
- Node-based
  - Uncontrolled string
  - Controlled string
  - [English word root] ... we will not build this semester.
- Context-based
  - Text
  - Structured document

Link-Based Searchers

Description: link class + direction + pattern description

Algorithm

- The “pattern description” describes a (set of) nodes, possibly in a further context of other links and nodes
- Other searcher(s) will map it to a WtdObjSet of nodes that may occur at the appropriate end of links in this class
- This searcher then retrieves set of links in the class that terminate in each of the nodes in the WtdObjSet
- Finally, the sets are merged to form a union set.

Links in MARIAN may be absolute or weighted.
Absolute links produce flat link sets.
Weighted links produce weighted sets.

All link searchers in current MARIAN systems use a maximizing union.

Node-Based Searchers

Description: Either an ID or
Any object (in current systems, a String).

Algorithm:

- Consult local databases.
- Uncontrolled string:
  Check for ID.
- Controlled string:
  Check for ID or
  Exact match on string
- English word root
  Apply successive English morphological transformations.
  At each step, check for exact match on string.
  Weight the matches by transformational distance.

Context-Based Searchers

Description: node description + (link description)*

Algorithm:

- The (node description) and each (link description) each produce a WtdObjSet of matching nodes in this class.
- Merge the sets.

Searchers for both text and structured documents (all context-based searchers in current systems) use a summative union.

Searcher Operations

Types:

Maximizing Union

Summative Union

[Arbitrary Function Union]

... which we have no need for yet and will not build this semester.

[Absolute Intersection]

... which we have no need for yet and will not build this semester.

But they’re part of the design space.

Considerations:

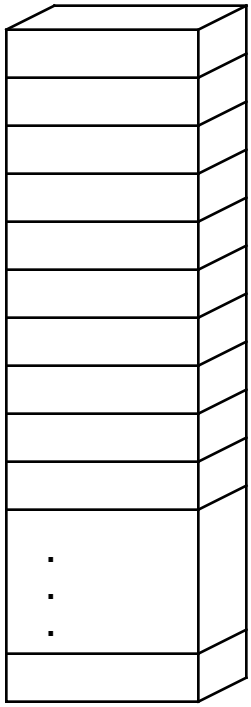
Number of sets to be combined.

Average size of set.

Are component sets proper WtdObjSets
or are they “flat”?

How precise do result set weights need to be?

Searcher Components: FullID Tables



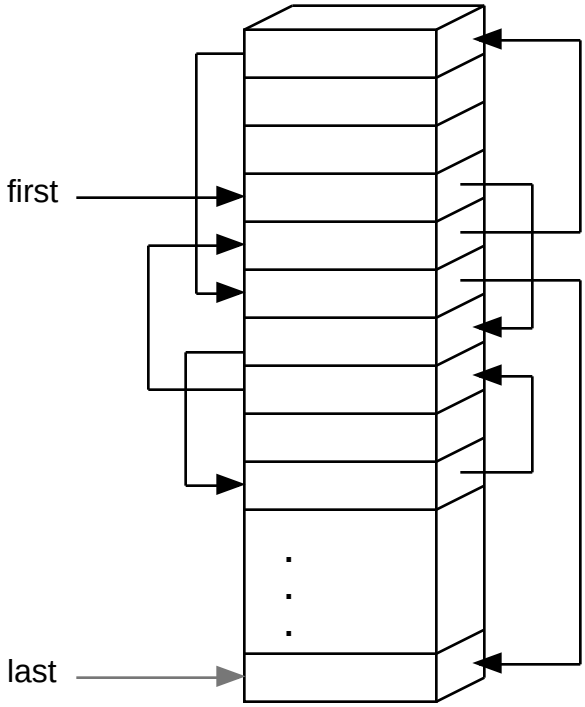
A FullIDTable is a hash table of objects, keyed by FullID.

The simplest FullIDTable is just a hash table of FullID objects.

A FullIDStringTable is a FullIDTable where the objects are pairs ‹FullID, String›.

A WtdObjTable is a FullIDTable where the objects are WtdObjs (FullIDs + Weights).

An AbritraryFullIDTable contains objects of class Object. To enable threading, however, the FullID of each object must be recoverable, so the object are actually pairs <FullID, Object>.

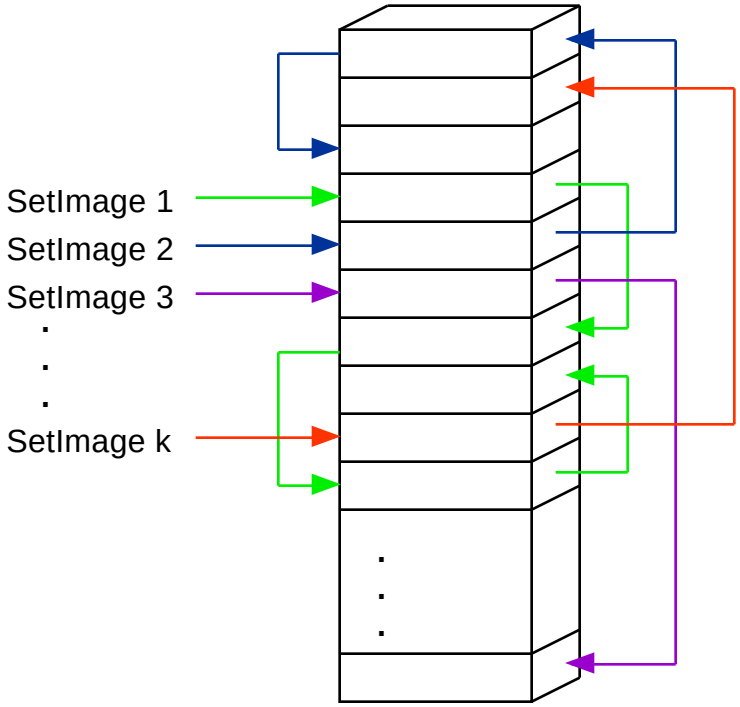


Most of the FullIDTables we use to construct Searchers are actually *threaded* (in the linked list sense, not the process sense). In other words, in addition to whatever objects are stored as data, the hash table objects also include either single or double links, allowing us to connect them in orders that have nothing to do with the table construction.

In a singly threaded FullIDTable, the objects in the table form a singly linked list.

An InsertionOrderedFullIDTable maintains a single list through the objects in the order in which they are stored.

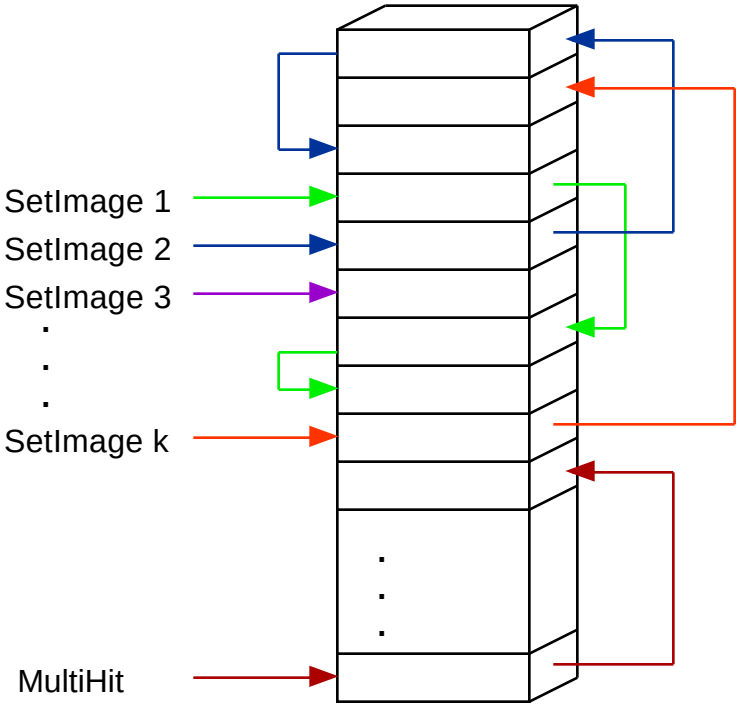
The maximizing union searcher, for instance, uses an InsertionOrderedWtdObjTable.



In a multiply threaded FullIDTable , the objects in the table form several linked lists. As in the singly threaded case, several list disciplines are possible. We will be particularly interested again in insertion ordered lists.

The primary use for this table is to maintain a collection of *set images* for searchers that need to examine all elements in each of the component sets, but may not need to put them all into order.

In particular, Summative Union and Arbitrary Function Union Set searchers maintain doubly linked lists for the elements of each component set that occur in no other set. Typically, these images comprise most of each set.



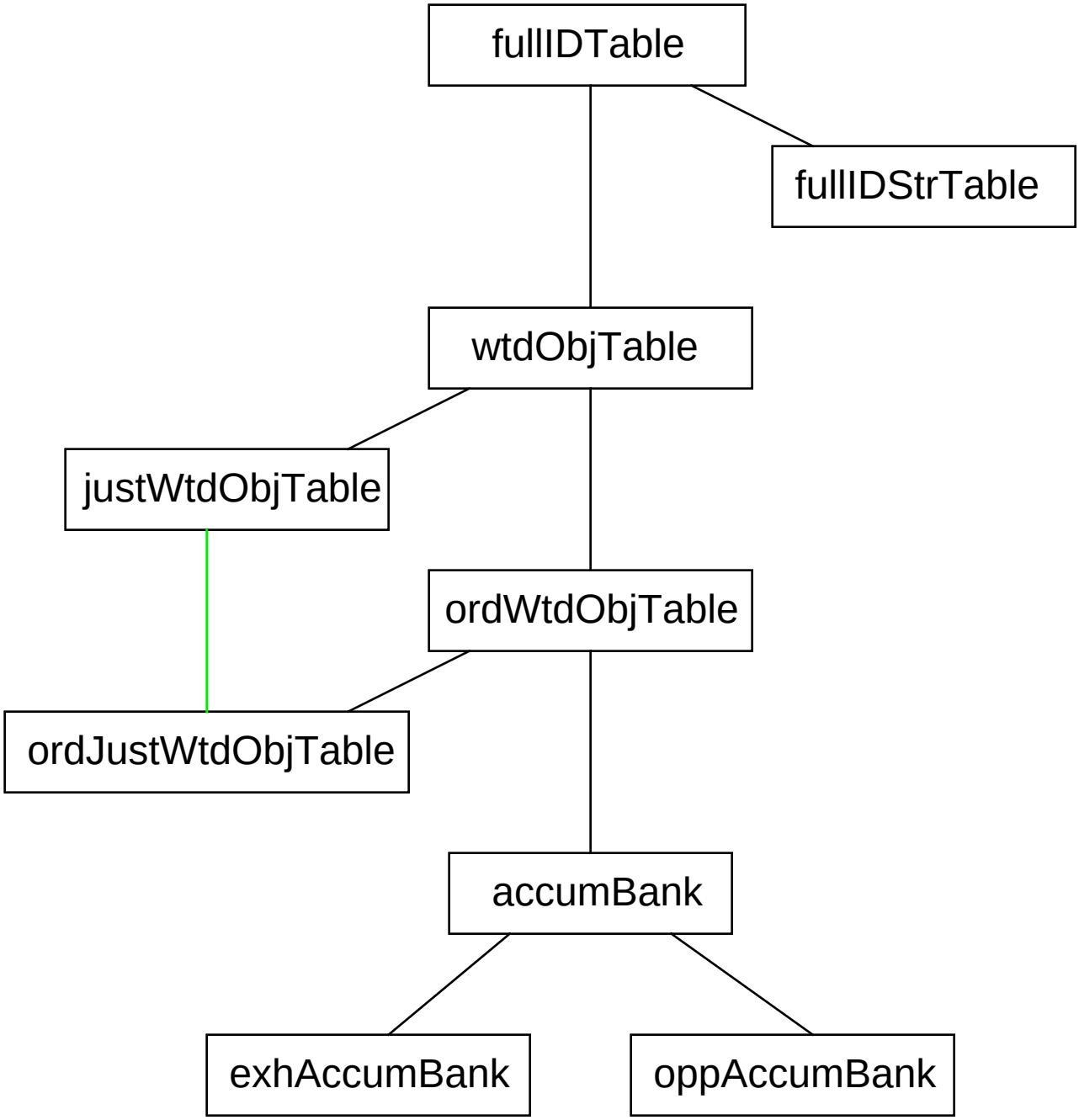
As each new element is read from a component set, it is deposited in the table.

If the element has never been seen before (its FullID is not yet in the table), it is hooked to the end of its own set image.

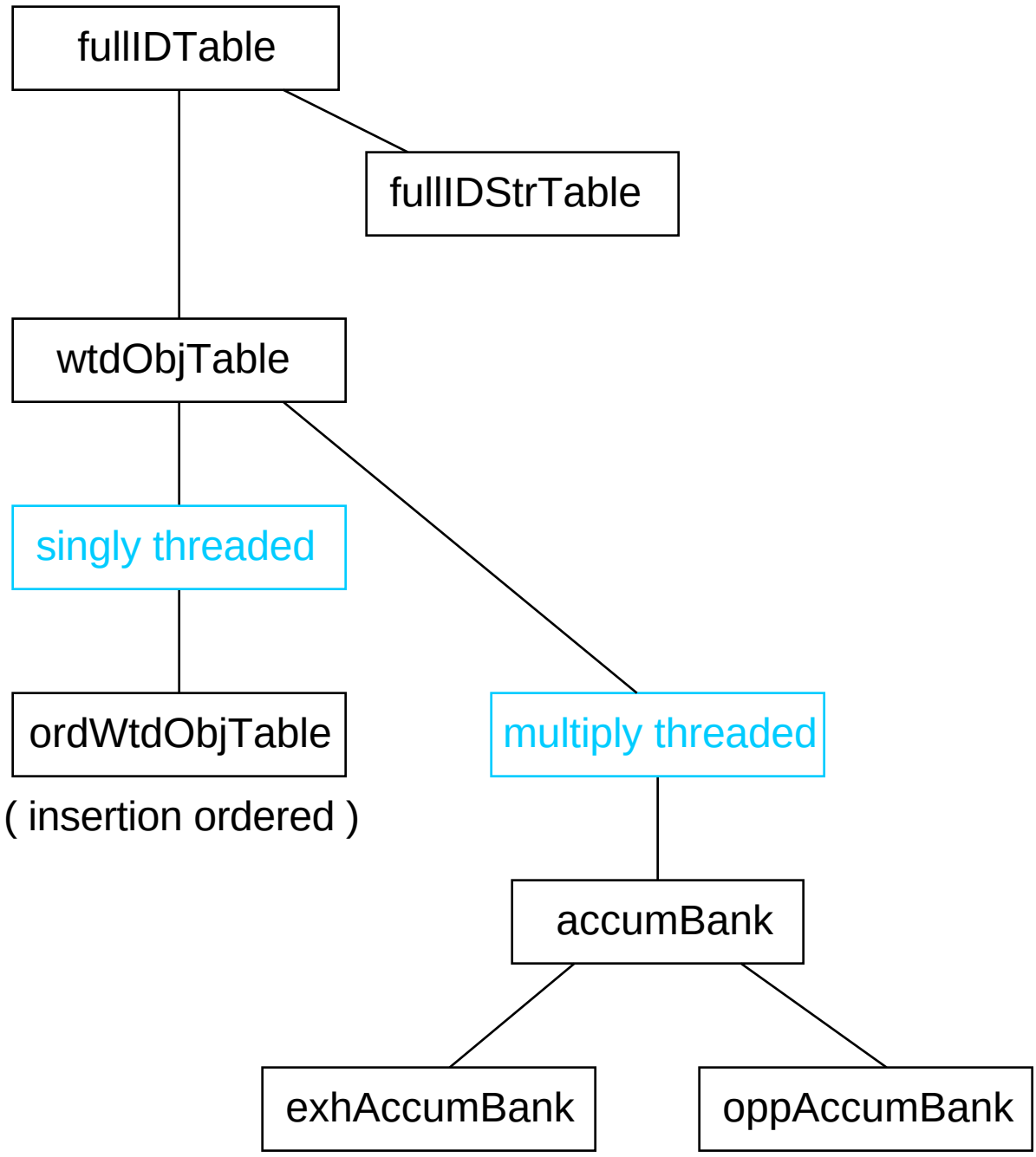
If the element is already in the table in another set image, it is detached from that list and attached to a list of “multiply hit” objects.

If the object is already on the multiply hit list, it is left there.

Whether there is one multiply hit list or several, and whether it is (or they are) maintained in order or kept unordered until the searcher is finalized, depends on the particular searcher implementation.

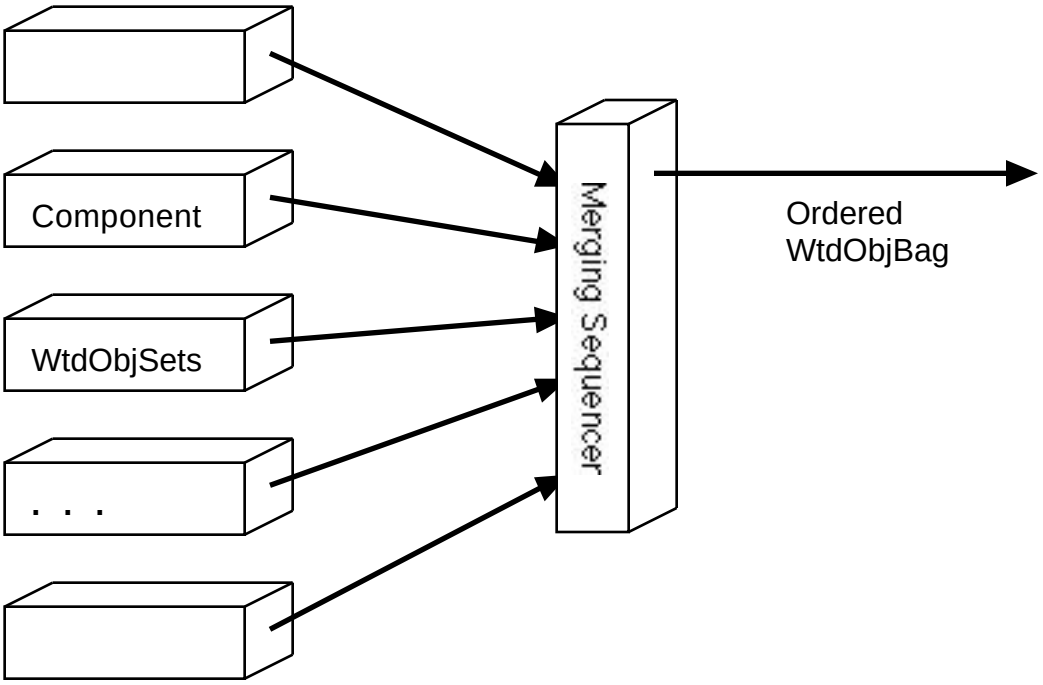


FullIDTable classes as they were implemented in C++ MARIAN.



FullIDTable classes as they will be implemented in Java MARIAN.

Searcher Components: Sequencers

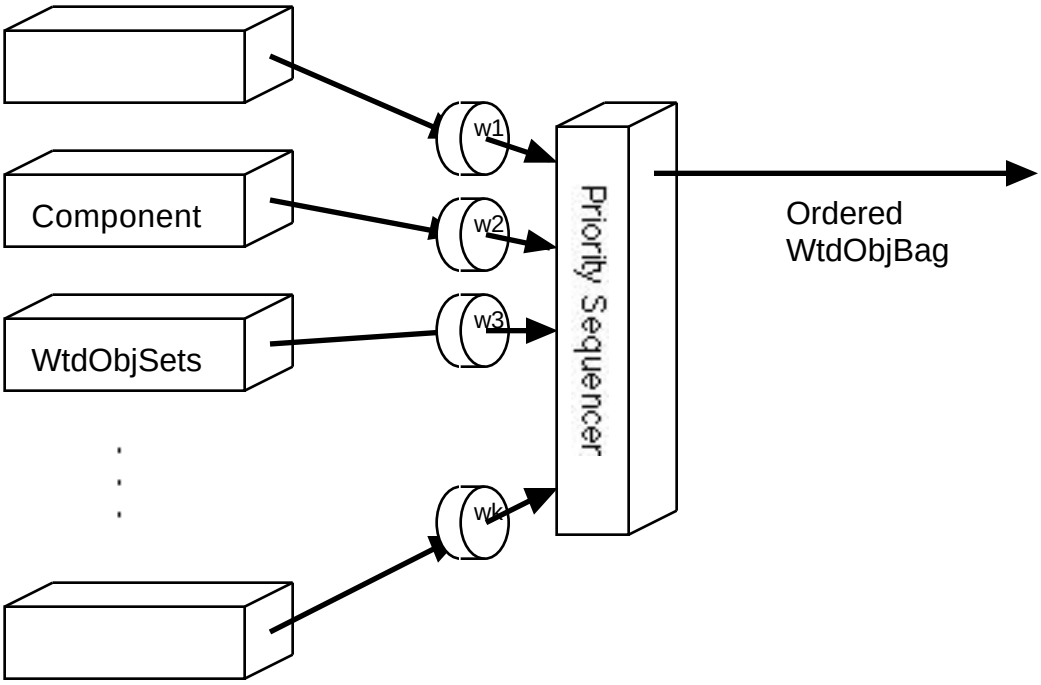


A Sequencer is a device for merging WtdObjSets without attempting to identify common elements.

In the most common application, a small number of proper WtdObjSets, each with a large number of elements, are to be merged.

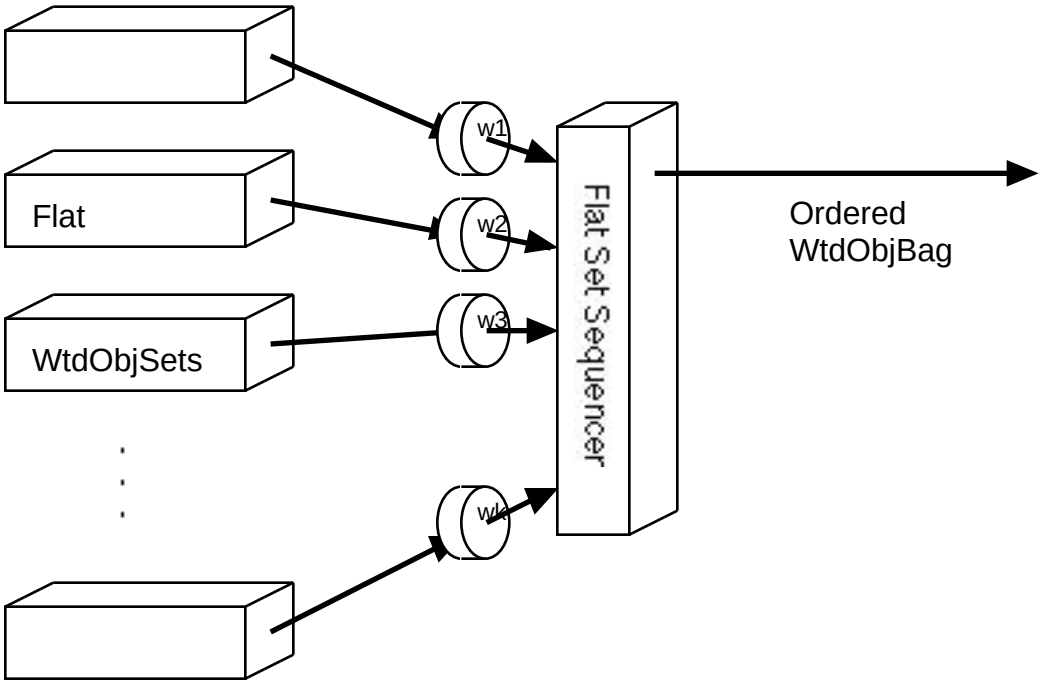
In this case, we get best performance by comparing the top elements of all component sets each time a new element is requested.

If the sets have significantly disparate weights, or if (as often happens) the component set functions are “stairstep” functions, we can improve performance by always remembering the list with the next highest unused element. Then we only need to scan for (second) highest weight each time we change lists, instead of each time we output an element.



We also encounter the situationwhere there are a very large number of component sets. This occurs most frequently when, as in link searchers, one must process a weighted object set of weighted object sets.

This operation requires more effective means – as for instance a priority queue – of choosing the next set from which to take elements.



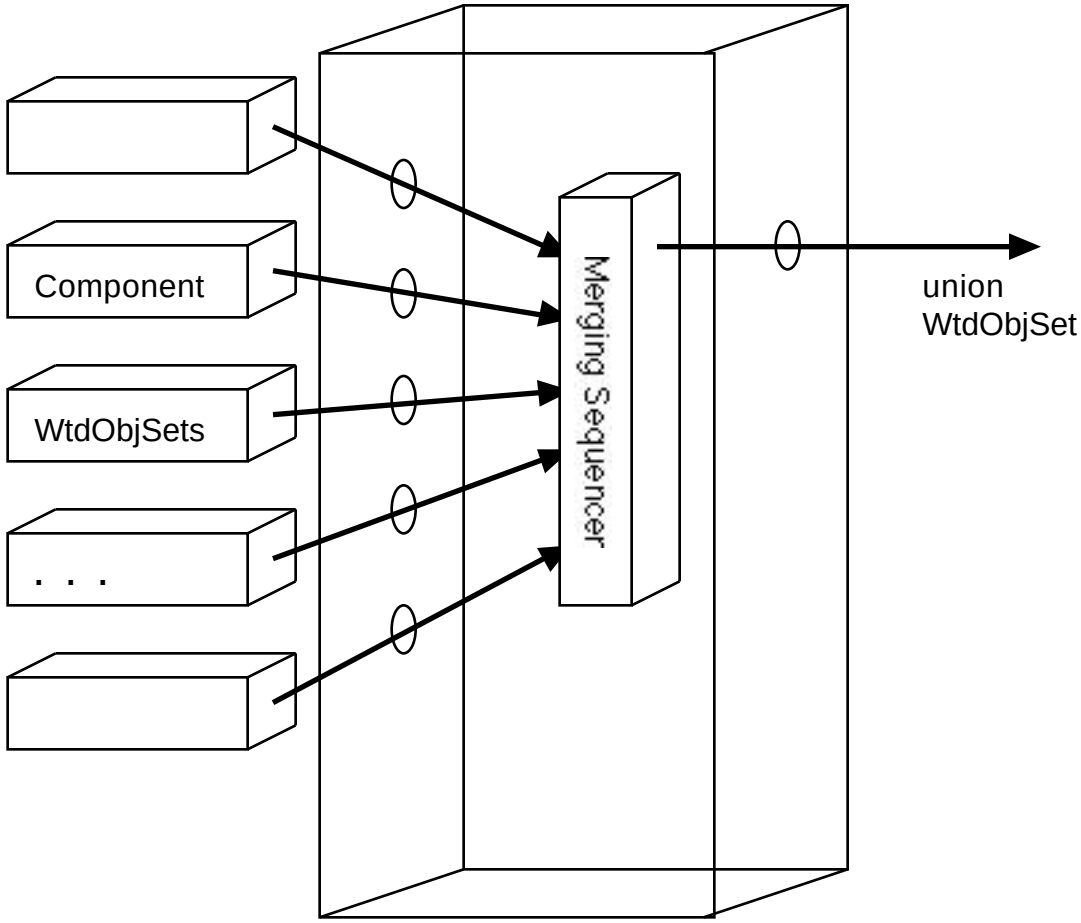
Finally we encounter the case for absolute (unweighted) links. Here the component sets are all “flat,” with no variation among their values. Moreover, all the objects in all the component sets have the same value. Thus the only weighting (and thus the only ordering) imposed on the elements in the collection is that imposed by the constnats c1, c2, ..., ck (which is to say, the weights in the outer weighted object set). The componenet sets, once established, acn be explored in the order given by the outer set. In fact, none of the component sets need even be established until the enumeration reaches them.

Consider the

nodeSetToNodeSet()

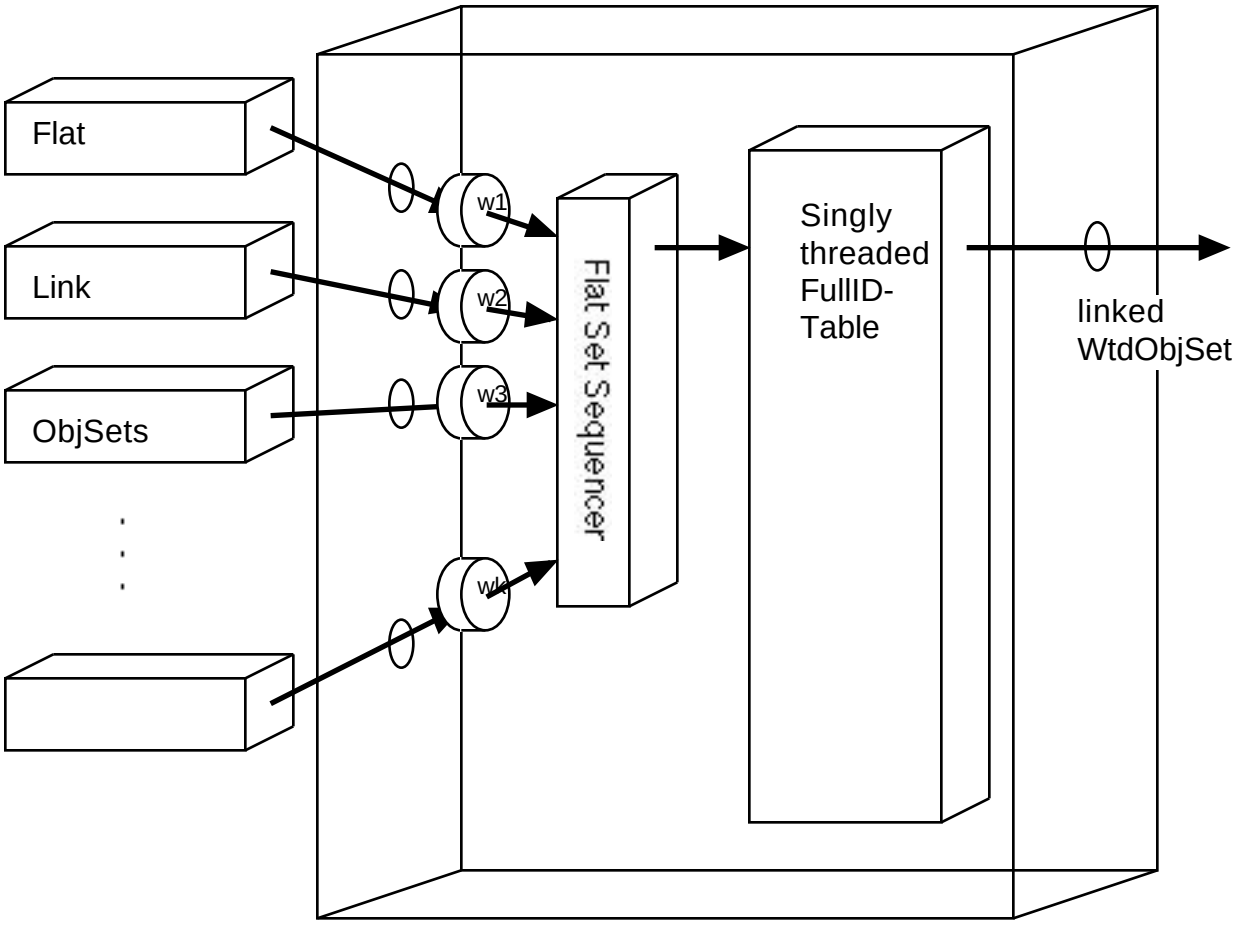
method of an unweighted link searcher. Each object in the key set of source or sink nodes can be used to retrieve a set of sink or source nodes respectively. But retrievals need only be performed as they are needed to supply requested elements.

Parent class merging searcher.



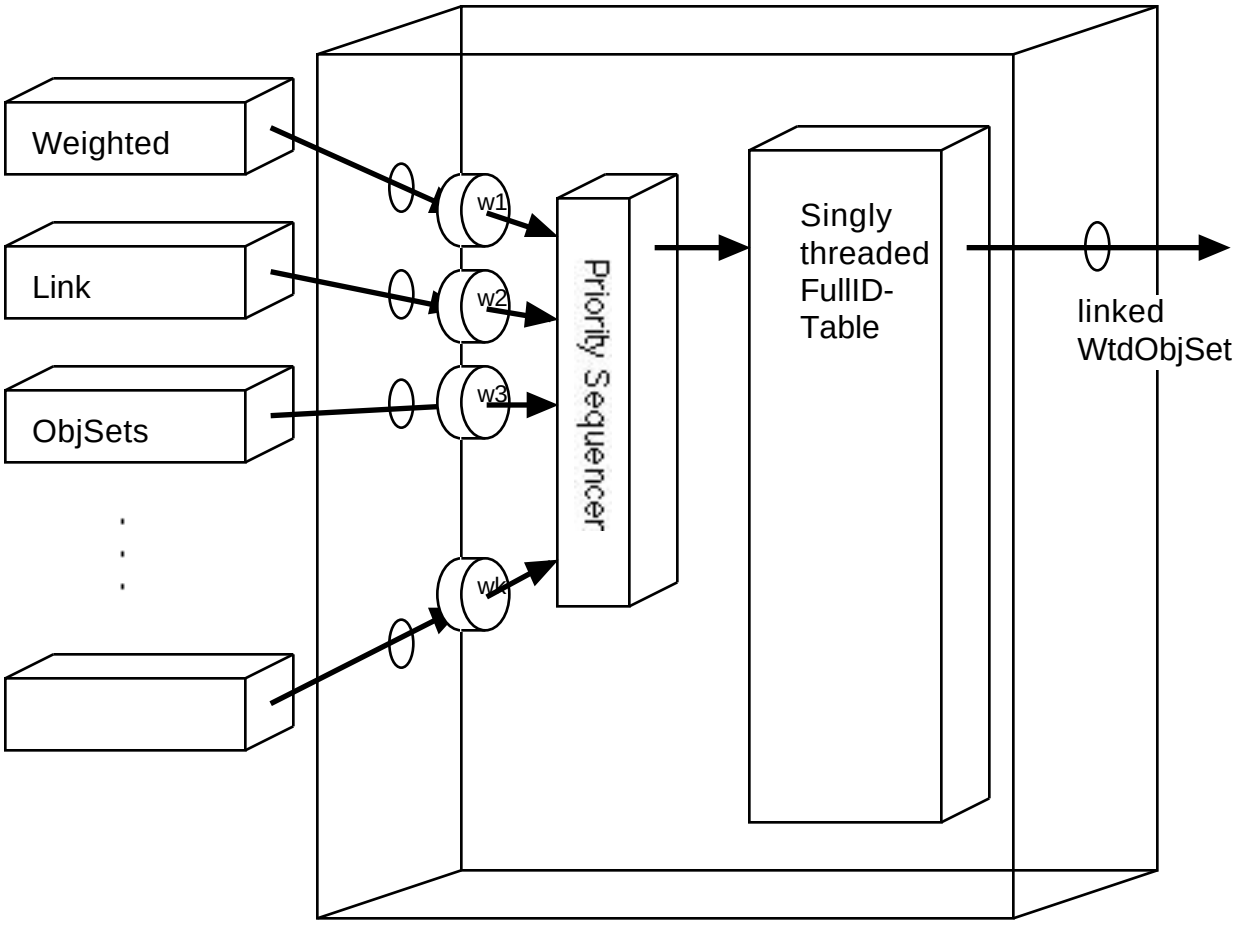
Implements: disjoint weighted object set union.

has Attribute link searcher.



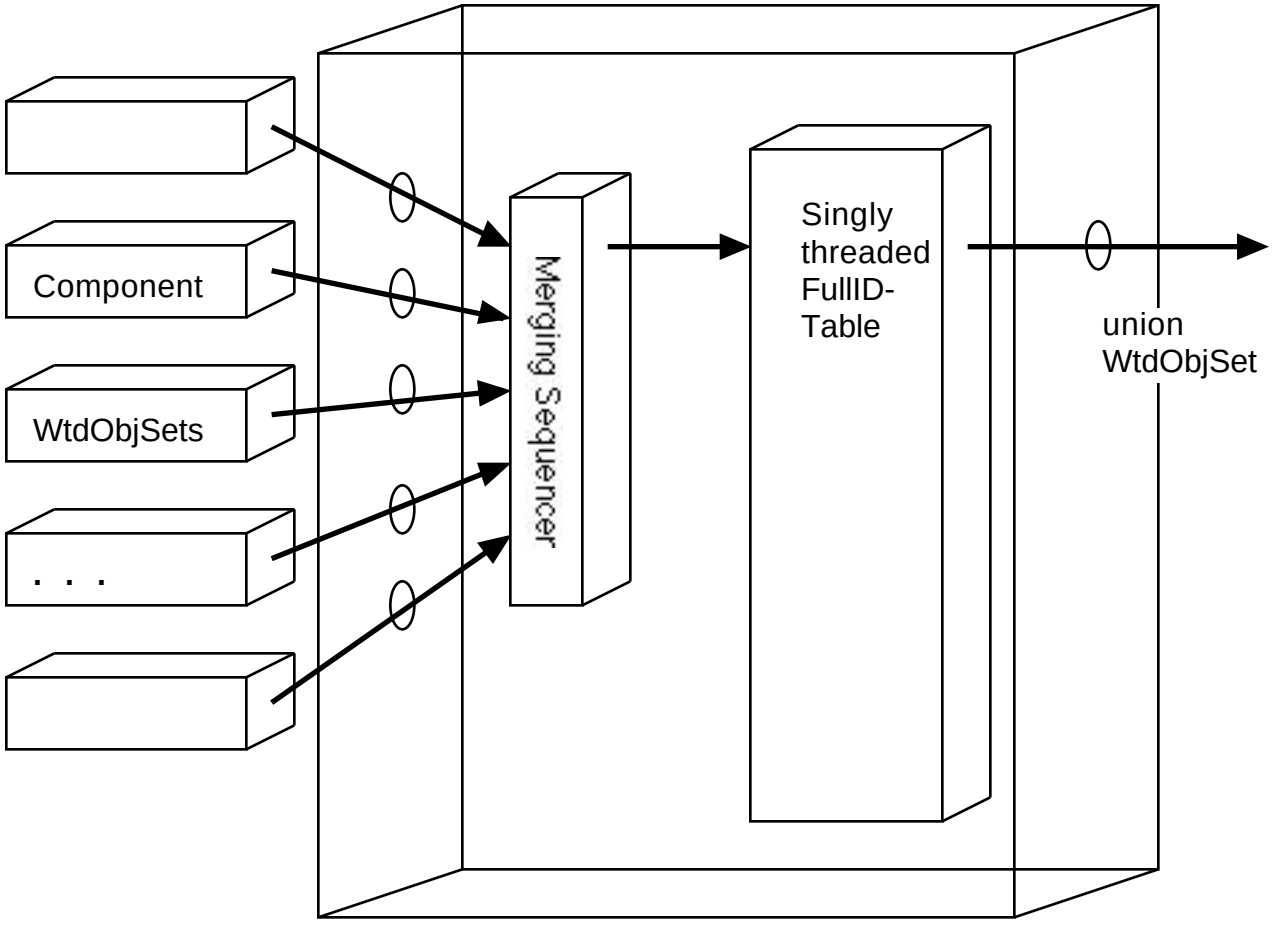
Implements: maximizing union for a (potentially large) weighted set of flat (unweighted) object sets.

occurs In **link searcher.**



Implements: maximizing union for a (potentially large) weighted set of weighted object sets.

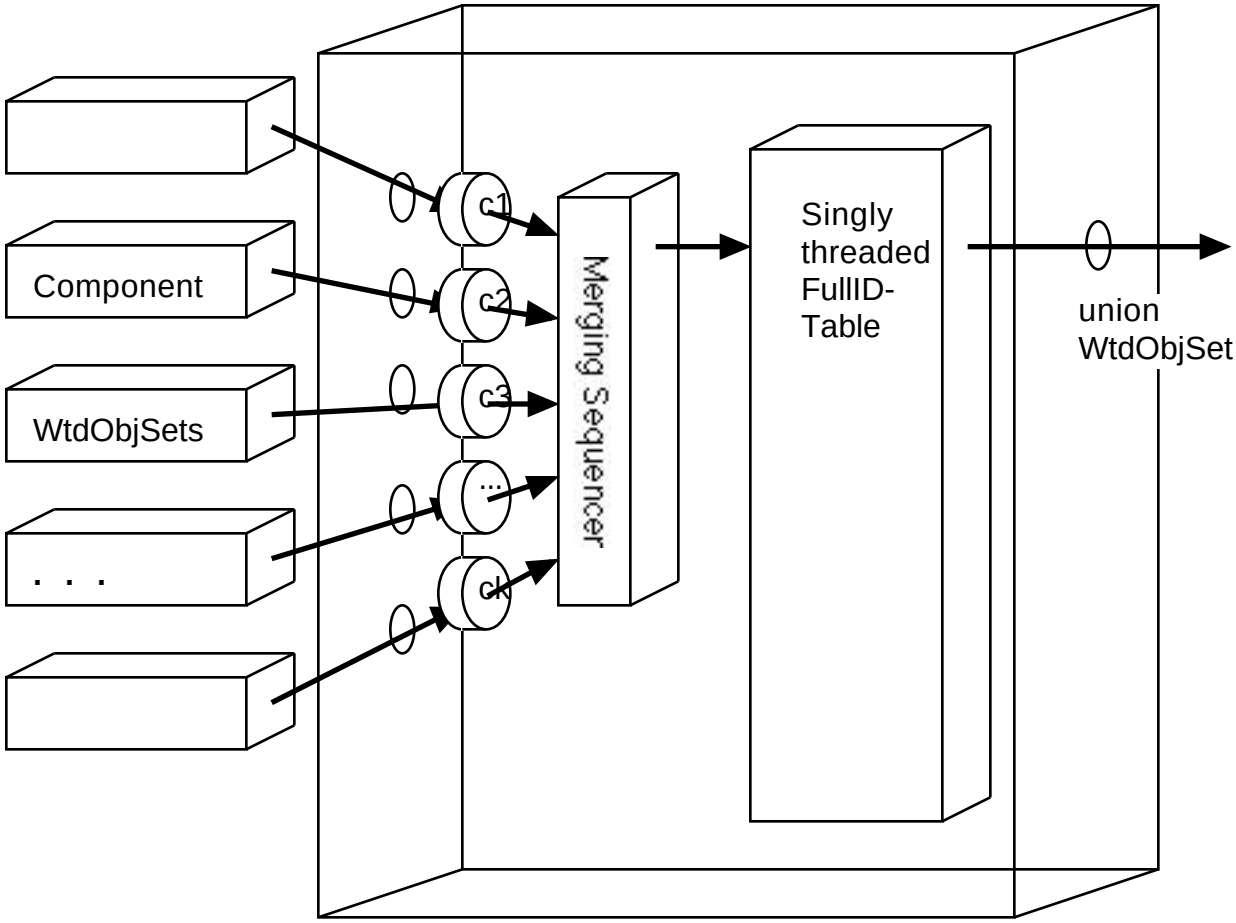
Lombard: A very simple union searcher.



Implements: maximizing weighted object set union.

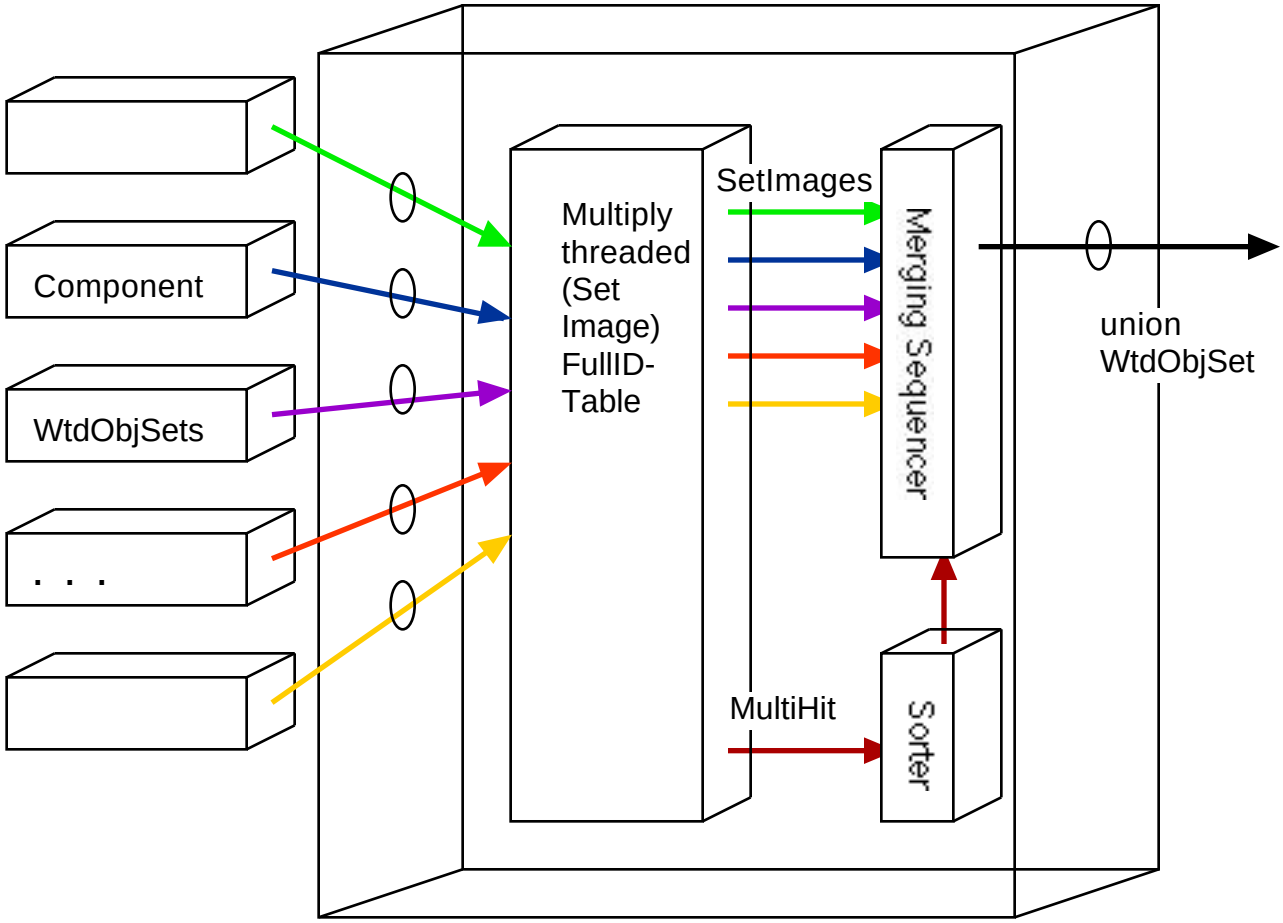
Algorithm: Get elements from component sets in globally non-increasing weight order.
If element not already in Table, add to Table at end of thread.
Repeat only as needed to fulfill requests on the union WtdObjSet.

Restrictions: Assumes component sets already in order.
Assumes there are few enough components that finding the next set from which to draw elements is an **O**(few) operation.
Sequencing operation is needed only when component sets are not “flat.”



No algorithmic complexity is added if the component sets are scaled by constants, as happens when they are themselves derived from a weighted set of components.

Acquinas: An exhaustive summative union searcher.

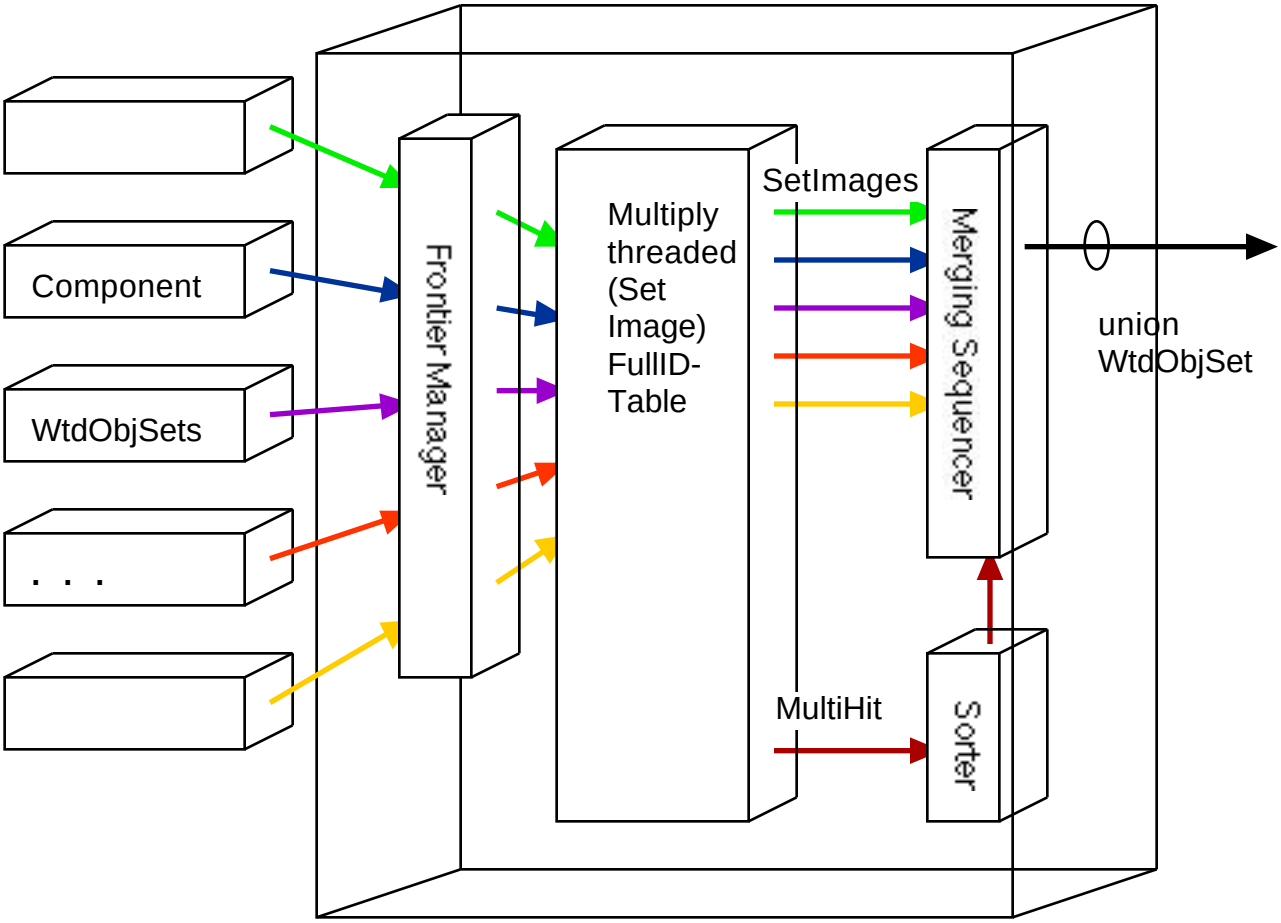


Implements: summative weighted object set union.

Algorithm: Run each component set into a linked *set image* in the Table.
If element not already in Table, add to Table at end of image.
If element is in Table, it must also be in some other component set.
Detach it from its current image and attach it to MultiHit,
adding the component set weight to its current weight.
When all component sets have been converted to images,
sort MultiHit.
Return elements from MultiHit or remaining (now disjoint) images
as requested by client.

Note: Most (though not necessarily all) of the elements of MultiHit will have higher weight than most of the singly hit elements in the set image remainders. Therefore the sequencing operation is trivial for the first segment of the union set.

Occam: An opportunistic summative union searcher.



Implements: summative weighted object set union.

Algorithm: Estimate a weight below which it is unlikely that combinations of elements will be in the union set segment requested.
Run all elements above that *frontier* weight from each component set into the set's image in the Table.
Promote elements to MultiHit as in Acquinas.
Sort and/or sequence the elements in the requested segment.
Re-calculate the frontier weight using this better data.
Repeat the process until the probability of a new element being added to the segment requested is below some tolerance level.

Note: For Occam's performance to be acceptable, we must be tolerant of some inaccuracies in the final weights.
Applications where reasonable tolerance levels can be achieved before all component sets have been completely run in are few, so we are not going to build this one in Fall '99.