Sweep And Prune

jodavis42@gmail.com

What's good changes over time

SAP used to be hot

Now it's not

Non-Uniform Grids

Uniform grids have problems

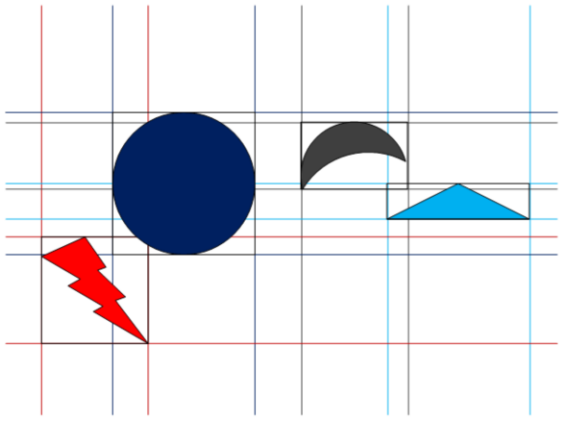
- Cell size
- Cell boundaries

Can we make a non-uniform grid?

For all of the benefits given by uniform grids so far they have a small set of big problems. The two main problems are what size cells to pick when we have varying sized objects and how to deal with objects that cross cell boundaries. The first of these problems are addressed by H-grids but not without cost. H-grids take a significantly larger memory footprint and queries are more costly. These are all problems because we've made a fixed grid in space independently of where objects are.

This leads to the question, can we make a non-uniform grid?

Aabbs Define a Non-Uniform Grid



Each object's Aabb defines a "cell"

The short answer is yes, we can define a non-uniform grid. In fact, aabbs implicitly define a non-uniform grid as shown above.

Aabbs Define a Non-Uniform Grid

Knowing this, how do we check for overlaps?

- Simple method is brute force: $O(N^2)$
- Too slow!

Key Observation: Aabb's are separable

- Each axis' overlaps are independent

The simple way to check for overlaps in a non-uniform grid is brute force which is n^2 complexity, but we can do much better than this!

The key observation that will provide improvements is that aabb intersection tests are separable, that is each axis' overlap result can be computed independently and combined later.

Aabb Overlap Separation

Basic (and slow) idea is to separate each axis' check

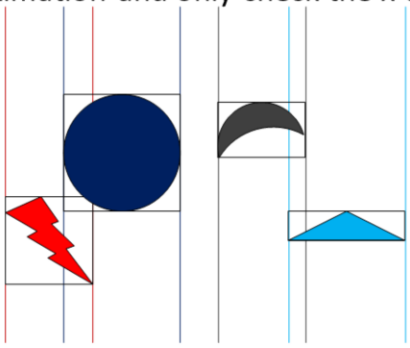
```
for(size_t i = 0; i < mAabbs.size(); ++i)
{
  for(size_t j = i; j < mAabb.size(); ++j)
  {
    // Check the aabbs for overlap
    CheckOverlap(mAabbs[i], mAabbs[j]);
  }
}
```

```
for(size_t axis = 0; axis < 3; ++axis)
{
  for(size_t i = 0; i < mAabbs.size(); ++i)
  {
    for(size_t j = i; j < mAabb.size(); ++j)
    {
      // Check the overlap only on the given axis
      CheckOverlap(mAabbs[i], mAabbs[j], axis);
    }
  }
}
```

The most basic idea here now is instead of doing a double loop over the input aabbs we now iterate over each axis independently and check for overlaps on that axis (a range check). So far this doesn't give us anything except more work but we'll slowly get somewhere better.

Simple 1-Dimensional SAP

Let's make an approximation and only check the x-axis (for now)

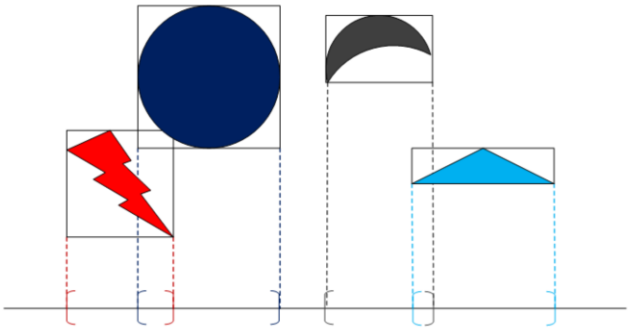


This will return (big) false positives but allows us to build up to something

As a spatial partition's job is to find false positives we can make a very crude approximation: If the aabbs don't overlap on the x-axis then we prune them out of our possible pairs. This means we can still return objects that overlap on the other axis' and leave that to narrow-phase to detect (for now). While this may save us some small work we still have algorithmic complexity of $O(n^2)$, but now that we're investigating one axis we can more easily solve this problem.

Simple 1-Dimensional SAP

How do we represent each object?



Min/Max projections on each axis (end points)

```
class SapBox
{
  void* mClientData;
  Aabb mAabb;
}
```

Each object has an associated box

```
class EndPoint
{
  bool mIsMin;
  float mValue;
  SapBox* mBox;
}
```

Each box creates two end-points for an axis

Moving forward to optimize this we need to first figure out how we want to represent each object on an axis. While there are other options I'll just jump forward to the simple representation of an end-point.

Simple 1-Dimensional SAP

How do we optimize one axis?

Turn $O(n^2)$ pair checking into 2 steps:

1. Sort endpoints by value $(n * log(n))$
2. Do a linear pass to find pairs (n)

Total (basic) complexity is $n * log(n)$

```
class SapBox
{
  void* mClientData;
  Aabb mAabb;
}
```

```
class EndPoint
{
  bool mIsMin;
  float mValue;
  SapBox* mBox;
}
```

So with this collection of end-points on each axis how can we optimize finding pairs? Well I'm going to claim that we can turn finding pairs from $O(n^2)$ into a sort operation (technically $O(nlogn)$ but we'll come back to this) and a linear sweep.

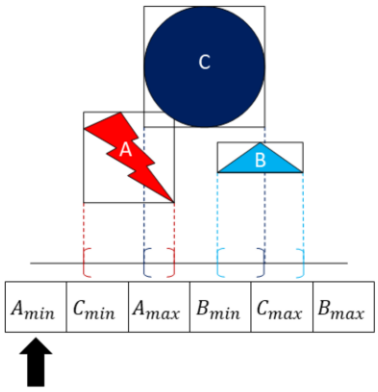
Simple 1-Dimensional SAP

Finding pairs on a sorted array is simple:

```
for(size_t i = 0; i < mEndpoints.size(); ++i)
{
  if(mEndpoints[i].IsMin())
  {
    // Create pairs with everything in our map
    // Add this object to the map
  }
  else
    // Remove from the map
}
```

The algorithm to find pairs on a sorted array is simple with the use of an associative container. Simply walk over all end-points, and add/remove them to a map depending on if they're a min or a max.

1-D Sweep Example



EndPoint Set

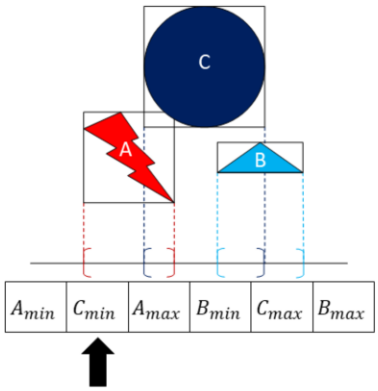
A

Registered Pairs

Encountered a min: Create pairs with everything in the set and then add to set

Now for a quick example of performing this sweep to find pairs. To start we have A's min. There's nothing already in the end-point set so no pairs are created. Object A is then added to the set.

1-D Sweep Example



EndPoint Set

A, C

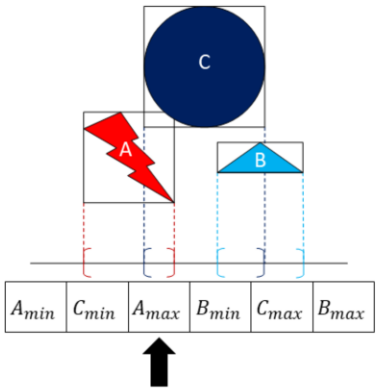
Registered Pairs

(A, C)

Encountered a min: Create pairs with everything in the set and then add to set

No we find C's min. Since A is already in the set we create the pair (A,C) and then add C to the set.

1-D Sweep Example



EndPoint Set

C

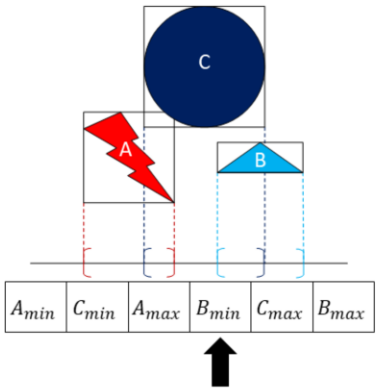
Registered Pairs

(A, C)

Encountered a max: Remove from endpoint set

Now we find A's max. As this is a max end-point we just remove the associated object from the end-point set.

1-D Sweep Example



EndPoint Set

B, C

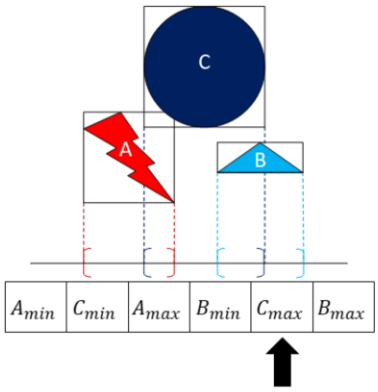
Registered Pairs

$(A, C); (B, C)$

Encountered a min: Create pairs with everything in the set and then add to set

Once again, we find B's min so we create the pair (B,C) and then add B to our set.

1-D Sweep Example



EndPoint Set

B

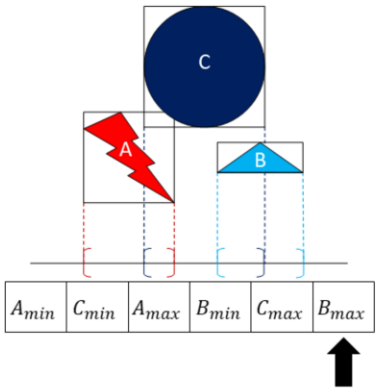
Registered Pairs

$(A, C); (B, C)$

Encountered a max: Remove from endpoint set

C's max removes C from the end-point set.

1-D Sweep Example



EndPoint Set

Registered Pairs

$(A, C); (B, C)$

Encountered a max: Remove from endpoint set

Finally we have B's max so we remove B from the set.

Simple 1-Dimensional SAP

Sorting can be achieved with your favorite algorithm

I recommend Insertion Sort for reasons I'll get to later

Now for one axis we have $O(nlogn) + O(n)$ complexity

Another name for Sweep and Prune is Sort and Sweep

Sorting on an axis is simple and can be achieved with any sorting algorithm you know of. This technically will gives us an $O(nlogn)$ sort time with something like quick-sort, but for reasons I'll get to soon I actually recommend insertion sort even though it is $O(n^2)$.

Now we've finished off a 1-dimensional SAP algorithm, or to be more precise this version of SAP is often called Sort and Sweep instead. This should be obvious from the name: we perform a sort over an array and then a linear sweep to find pairs.

N-Dimensional SAP

Shapes only overlap when all axes overlap!

- Sort all axes
- Sweep over all axes, only inserting a pair if all overlap

Now that we've established how SAP works in 1-Dimension it shouldn't be too hard to see how we can extend it to 2 or even n-dimensions. Just as with an aabb where we combine each axis' results and only register a collision if we have an overlap on all axes, with SAP we simply keep track of how many axes a pair is overlapping on. If there is overlap on all n axes then the shapes overlap.

One alternative to SAP is to only sort and sweep on 1 (or more) axes and perform the remaining aabb check before inserting a pair.

Simple SAP Structure

We can now define the most basic SAP structure:

```
class Sap
{
  Array<SapBox> mBoxes;
  Array<EndPoint> mEndPoints[3];
}
```

```
class SapBox
{
  void* mClientData;
  Aabb mAabb;
}
```

```
class EndPoint
{
  bool mIsMin;
  float mValue;
  SapBox* mBox;
}
```

We'll build up to something better (and more complex) now.

Ask now if you are confused!!

*Please ignore any structure issues for now…

This is the basic data structure of what's been described so far. Our SAP class has a collection of boxes, one for each object. This box stores the client data inserted as well as the aabb inserted. Additionally we have several arrays of end-points (one for each axis). Each end-point array is sorted based upon the value in the end-point structure. To add/remove pairs we also store the box pointer in each end-point.

This structure has several problems (such as pointers into arrays), but ignore them for now. I'll start going over the canonical form of SAP now that we understand the basics of how everything works. As a warning, while this will give quite a big performance boost it will also make things much more complicated!

Spatial Coherence – EndPoint Sorting

Problem 1: Why are we sorting each axis every frame?

Object's don't move much from frame-to-frame!

Examples:

- What if only one object moves?
- What if no objects move?
- What if all objects barely move (endpoint order might not change)?

The first thing to note is that we are wasting a lot of time by sorting each axis from scratch every frame. If we have 100 objects but only 1 moves why not store last frame's sorted axes and just update the endpoints for that box? This is an extreme example but it comes up more than you might think. In particular, if we look at the principle of spatial coherence you'll see that this happens a lot! As objects barely move from frame-to-frame there's a good chance that even if every object moves that they've only moved a little and hence we'll have an update very close to n.

Spatial Coherence – EndPoint Sorting

Solution: Store last frame's endpoints and sort in-place

Sort algorithm matters a lot here!

- Quick-Sort is always $nlog(n)$
- Insertion-Sort is n when the array is already sorted

New (expected) complexity: $3(n + n)$

This leads us to the obvious solution of keeping around last frame's sorted endpoints. We have to be careful of our sorting algorithm though, as some of the "better" sorting algorithms like quick-sort are always (worst and best case) $nlogn$. This is where insertion sort will out-perform other sorting algorithms as it is n when the array is already sorted.

Now we can expect SAP to sort in almost n time for each axis plus as additional n per axis of finding overlaps

Spatial Coherence – Pair Registration

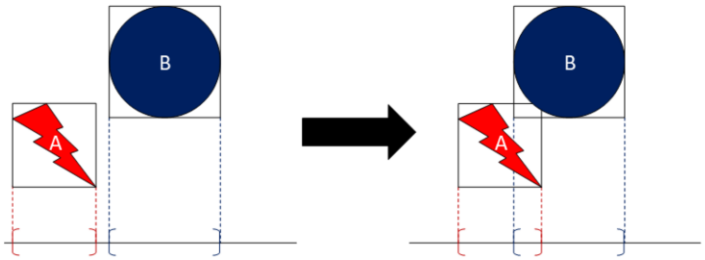
Problem 2: Why perform a full pass on an axis to find pairs each frame?

Idea: Cache last frame's pairs and incrementally update!

The second major optimization we can make is also related to spatial coherency. If we don't sort each frame then why are we finding all pairs every frame? Why not use last-frame's cached pairs and update them incrementally whenever we move (or insert or remove) an object?

Spatial Coherence – Pair Registration

4 Rules for updating:



1. A max becomes greater than a min: Insert a pair
2. A min becomes less than a max: Insert a pair

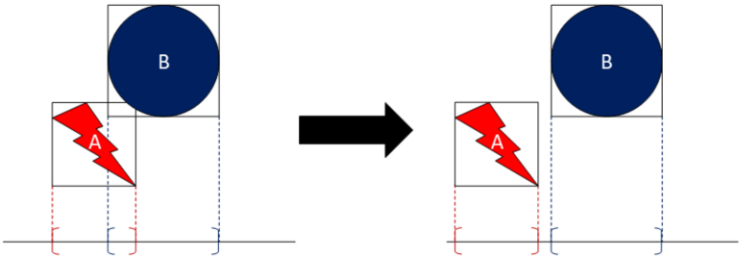
* Depends on if A or B was moving

It turns out we can build 4 simple rules (plus one minor rule) that allow us to determine whether to add or remove a pair while swapping end-points. This means while performing insertion sort we can simultaneously (un)register pairs.

The first two cases both are when a pair needs to be inserted. These cases are effectively the same thing, they only differ by which object moved into the other.

Spatial Coherence – Pair Registration

4 Rules for updating:



3. A max becomes less than a min: Remove a pair
4. A min becomes greater than a max: Remove a pair

* Depends on if A or B was moving

The second two cases both cause a pair to be removed.

Spatial Coherence – Pair Registration

No other cases matter (min > min, max < max, etc…)

*One exception: Make sure to update an object's min/max in the right order to avoid self-pairs

This will allow us to efficiently update pairs while sorting

Complexity $\approx 3n$

You might be wondering about all of the other end-point cases, but none of them matter as none of them can create/remove a pair.

There is one small exception to this rule that only applies to "fast" moving objects. If you aren't careful and a max of an object becomes less than its own min (because it's moving fast to the left) then a pair will be registered with itself. This is easy to fix by just adjusting whether which end-point is moved first. If the object is moving to the right then update max then min, otherwise update min then max.

With this in place our theoretical complexity is now $3n$ because we sort in approximately n complexity on each axis while simultaneously registering pairs on each axis. Before going over the actual issues that come up (such as worst-case counter examples) I need to revisit the structure of SAP and do some examples.

SAP Structure - Revisted

```
class Sap
{
  Array<SapBox> mBoxes;
  Array<EndPoint> mEndPoints[3];
}
```

```
class SapBox
{
  void* mClientData;
  Aabb mAabb;
}
```

```
class EndPoint
{
  bool mIsMin;
  float mValue;
  SapBox* mBox;
}
```

When updating how do we find the box?

Give the box pointers to each endpoint?

- Storing pointers in an array is bad
- Linked lists are slower

Store indices?

- Close, but we'll go a step further

There's a few major structural problems with SAP now that we have cache last frame's data. Namely how do we find each endpoint to update. If we have to search for it then we're back to an n^2 complexity.

We could try to store pointers into the endpoint array but this is all sorts of bad. Not only would we have to manage and update the pointers, but any time we resize the array all pointers could go bad. One solution is to instead store the endpoints in a linked list. This will actually be slower in the long run, mostly due to memory overhead and cache coherency.

If we're sticking with arrays then we could store indices instead of pointers. Indices are stable when an array is resized, but we have to do a lot of maintenance on them. This is very close to what we'll end up with, but we'll go one step further to help optimize performance, mainly cache coherency.

SAP Structure - Revisted

```
class Sap
{
  Array<SapBox> mBoxes;
  Array<EndPoint> mEndPoints[3];
  Array<int> mIndices;
}
```

```
class SapBox
{
  void* mClientData;
  Aabb mAabb;
}
```

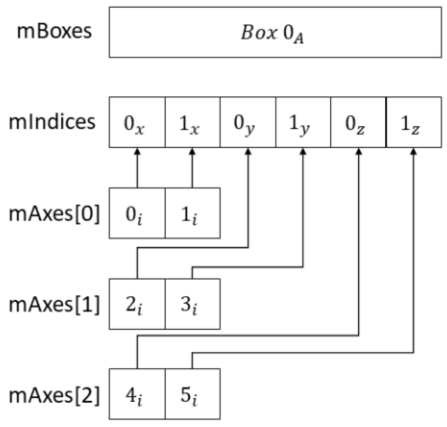
```
class EndPoint
{
  bool mIsMin;
  float mValue;
  int mIndex;
}
```

Give each endpoint an index (not into boxes)

Add this “weird” indices array

Now we’ve done some very minor changes for a much more stable (and efficient) SAP representation. In particular, we gave each endpoint an index (note: this is not the box index, I’ll get there later) and now the SAP structure has this weird array of indices. Understanding mIndices is crucial and requires pictures.

SAP Structure - Revisted



Indices is an in-between mapping of boxes to endpoints

1:6 box to endpoint mapping

Endpoint index points into indices

- Can check all endpoint values
- Can get to box by dividing by 6

Can hijack a bit from index for min/max flag

Very cache coherent

Questions!?

This weird indices gives us a 1:6 mapping of a box to its endpoints. Each box gets 6 entries in the indices array. Each of these indices corresponds to the locations of the min/max endpoints on each axis. Each axis now stores an index back into the indices array.

We can now very efficiently jump around from axes to indices to boxes to other axes and so on. Here's a quick list of what you can do:
Go from an endpoint to it's box: endpoint.mIndex / 6.
Go from a box to any min/max. For instance the y-min will be at index: boxIndex * 6 + 2
Compute a max's index for a min: mIndices[endpoint.mIndex + 1]
Go from any axis' endpoint to any other. For instance x-min to z-max will be at index: mIndices[endpoint.mIndex + 5].
And so on…

This is especially efficient as we can jump between all of these with as minimal cache misses as possible which leads to SAP being very fast. We can actually increase the cache coherency by a small amount by combining the min/max bool with the index (since we're never going to store 2 billion contiguous objects anyways…).

SAP

Insertion/Removal/Update are all very similar now

They all involve sorting endpoints into place

Now I can finally talk about insertion, update, and removal. They're all very similar though as they all have to sort end-points into place and register pairs. In fact, there's a high chance (read this as you should implement SAP like this) that they'll all share some function to sort end-points while registering pairs.

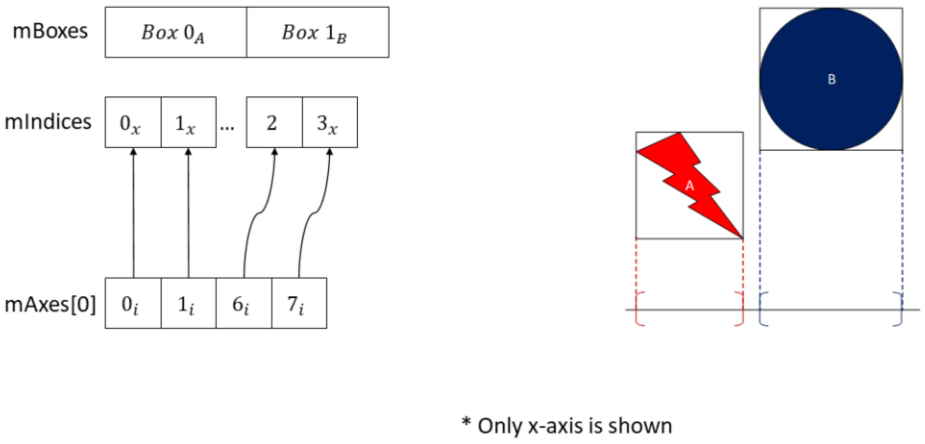
SAP - Insert

To insert:

1. Add a new box (or re-use an empty one)
2. Add 6 new indices (or re-use old ones)
3. Push 2 endpoints onto each axis and sort into place
   * Don't forget to register pairs while sorting

To insert we have to first determine if we have any boxes that are empty (that we're previously deleted). If so then we re-use that box and its corresponding indices. Otherwise we push back a new box and 6 new endpoints. In either case we then push back 2 new endpoints per axis and sort them into place.

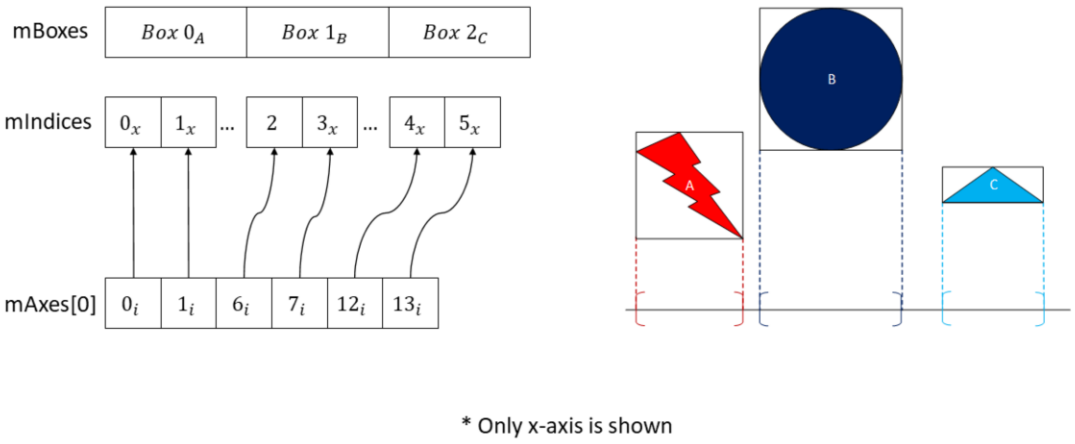
SAP - Insert



* Only x-axis is shown

Showing a very basic insert. We don't have any free boxes so we just push back a new one, 6 new indices, and push back 2 new endpoints. Note that only the x-axis is shown here for simplicity.

SAP - Insert



* Only x-axis is shown

Now we added shape C. Once again we just push back a new box, 6 new indices, and 6 new endpoints.

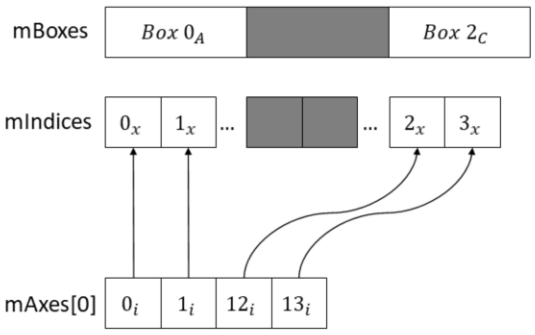
SAP - Removal

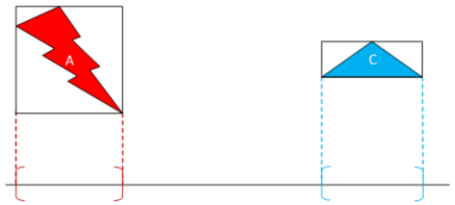
To Remove:

1. Mark the box as invalid
2. Set the endpoints to infinity and sort out
   *Make sure to remove pairs when sorting
3. Stop referencing the indices

To remove we simply mark the associated box as invalid and set all of the endpoints to infinity. After this running the sort update algorithm will move the endpoints to the end of each axis and we can remove them. This will also remove all pairs.

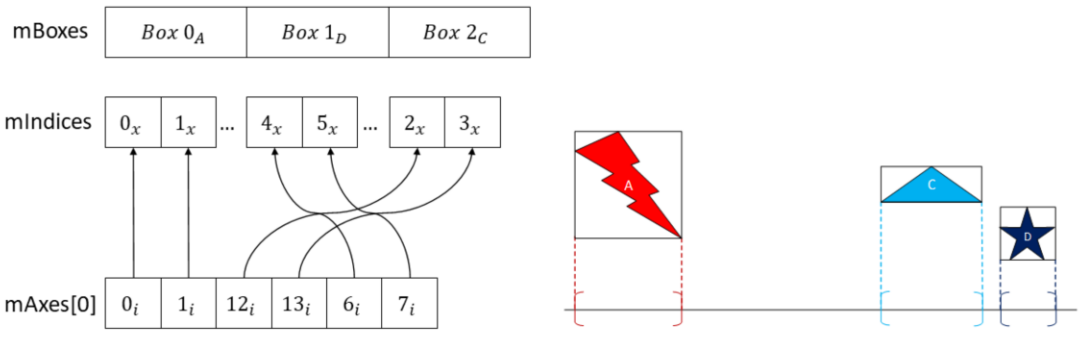
SAP - Removal





Now we deleted box B. We mark the box as invalid and sort the endpoints out to remove them.

SAP - Insert



Now we added another shape, but since we had a spare box we re-use it and its indices.

SAP - Update

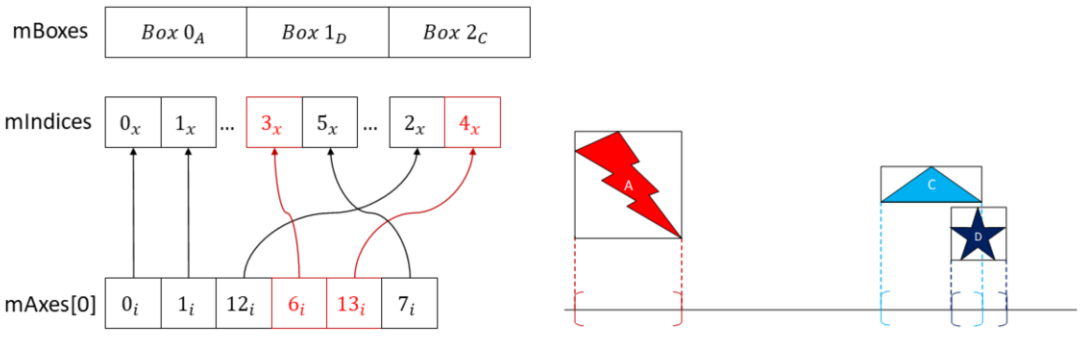
To Update:

1. Set the new end-point values and sort into place
2. Insert/Remove pairs via the update rules
   When there's an overlap on one axis check the remaining axes first
   Only insert a pair if all axes overlap
   * Due to geometric coherency, you only have to check indices

To update just set the new end-point values and sort them into place. We can then create pairs via the 4 update rules from before. The only difference is that since we are updating one axis at a time we need to check the remaining axes to determine if there's actually an overlap. We only insert a pair if all of the axis overlap.

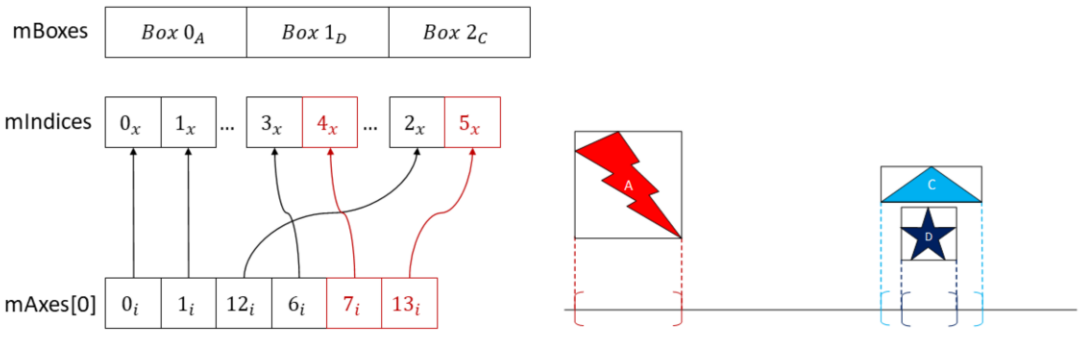
As a minor optimization, we don't actually have to check the aabb's values on the other axes. Instead, since all of the endpoints are sorted already we can check the indices of the other axes. If there is an overlap of the float values then logically there will be an overlap of the indices. This allows us to not only avoid floating point checks (slightly more expensive) but it also allows us to avoid indexing into the other endpoints which helps to avoid cache misses.

SAP - Update

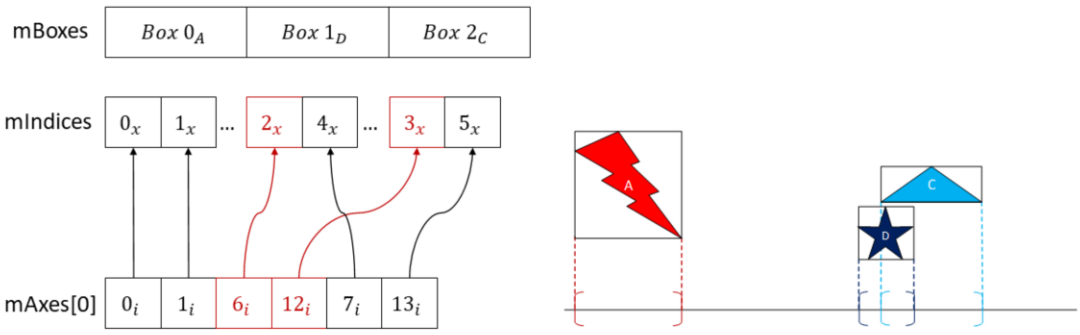


Now for a lot of update examples.

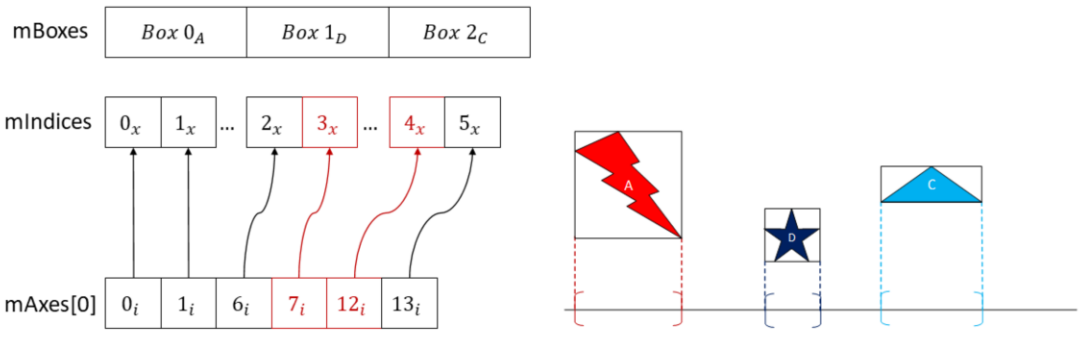
SAP - Update


mBoxes
$Box\ 0_A$
$Box\ 1_D$
$Box\ 2_C$
mIndices
0_x
1_x
...
3_x
4_x
...
2_x
5_x
mAxes[0]
0_i
1_i
12_i
6_i
7_i
13_i
A
C
D

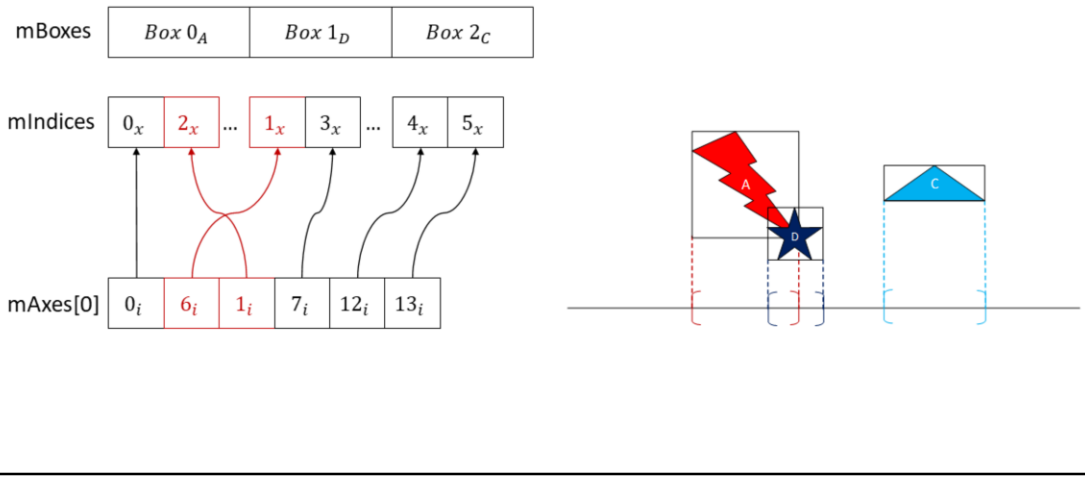
SAP - Update


mBoxes
$Box\ 0_A$
$Box\ 1_D$
$Box\ 2_C$
mIndices
0_x
1_x
...
2_x
4_x
...
3_x
5_x
mAxes[0]
0_i
1_i
6_i
12_i
7_i
13_i
A
C
D

SAP - Update


mBoxes
$Box\ 0_A$
$Box\ 1_D$
$Box\ 2_C$
mIndices
0_x
1_x
...
2_x
3_x
...
4_x
5_x
mAxes[0]
0_i
1_i
6_i
7_i
12_i
13_i
A
D
C

SAP - Update


mBoxes
$Box\ 0_A$
$Box\ 1_D$
$Box\ 2_C$
mIndices
0_x
2_x
...
1_x
3_x
...
4_x
5_x
mAxes[0]
0_i
6_i
1_i
7_i
12_i
13_i
A
D
C

Sentinels

Adding sentinel nodes make it easier to write the inner iteration loop

Be careful to not insert pairs with the sentinel

Adding sentinel nodes to SAP is very helpful. Sentinels clean up code to avoid out-of-bounds checks when updating nodes. The only thing to be careful about is during insertion/removal. In particular, during insertion new endpoints should be inserted before the last sentinel. During removal objects should be sorted out to some large value (like infinity * 0.5) instead to avoid creating pairs with the sentinels.

SAP Problems

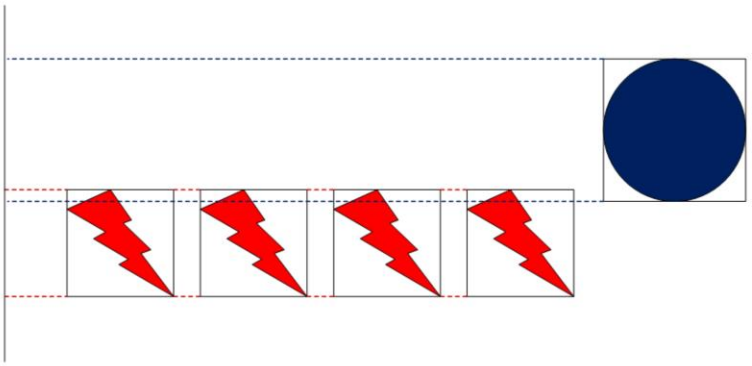
We rely on spatial coherency so that updates are cheap

Is there a scenario where a small update can cause a lot of change?

So far we've seen how SAP can find pairs very efficiently, namely it uses spatial coherency to only perform small updates. As long as an object doesn't teleport then it in theory will perform very few swaps. Unfortunately there are some edge cases where this isn't true…

SAP Problems - Clustering

Small changes can move across many endpoints



If there's clustering on an axis then SAP can perform very poorly. Since a small change can cause an object to move across many endpoints the update time can become very expensive. While this might not seem likely clustering tends to naturally happen on the y-axis if there is gravity and a flat ground.

SAP Problems - Clustering

Solutions?

- Ignore an axis
- Use non-axis aligned axes
- Multi-SAP

Unfortunately there's not many "great" solutions to this problem. The easiest is to not build/test an axis if we know it's likely to cluster. For instance, we could have a mostly flat game and choose to ignore the y-axis for SAP. This means objects overlapping on all axes except the y will incorrectly report a pair, but as long as there's not too many to worry about this might not be an issue.

Another way is to pick some non-axis aligned axes. Other axes can be less likely to have clustering, but unfortunately projecting onto arbitrary axes can often slow down the spatial partition enough to make this option less practical.

The final method to avoid clustering problems is to have multiple SAP spatial partitions in a uniform grid. This means that objects very far away will not cause swap orders with each other because they'll be in a different SAP. The practicality of implementing this is a lot harder, especially when dealing with an object straddling cell boundaries in the uniform grid. I'll defer more in-detailed talks of this to the Spatial Partition Extensions lecture, just know that this is an option.

SAP Problems – Massive Insertions/Removals

When an object is inserted/removed it can cross the entire array
If we do this for a lot of objects it's slow

Solutions?
Batch operations!

The other big problem that can happen is in insertion/removal. Since we detect pairs during the incremental sorting of endpoints we have to incrementally update when inserting/removing. This means that if we have a scene where a new object is inserted and its endpoints end up at the beginning of the array we have to sort into place (both to swap endpoints and to insert pairs). While this might not be too big of a problem for one off insertion/removals, if we perform several of these operations a frame we can see very large lag spikes. While there are several ideas on how to fix this, the only useful one to talk about are batch operations.

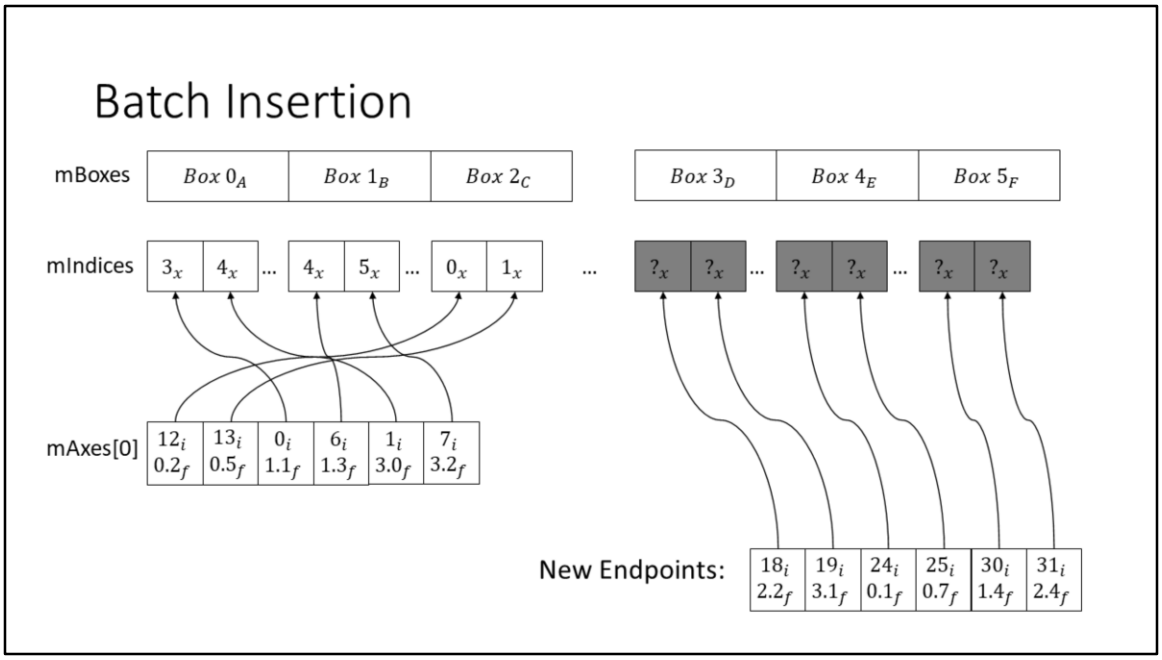
Batch Insertion

Endpoints are inserted in 3 steps:

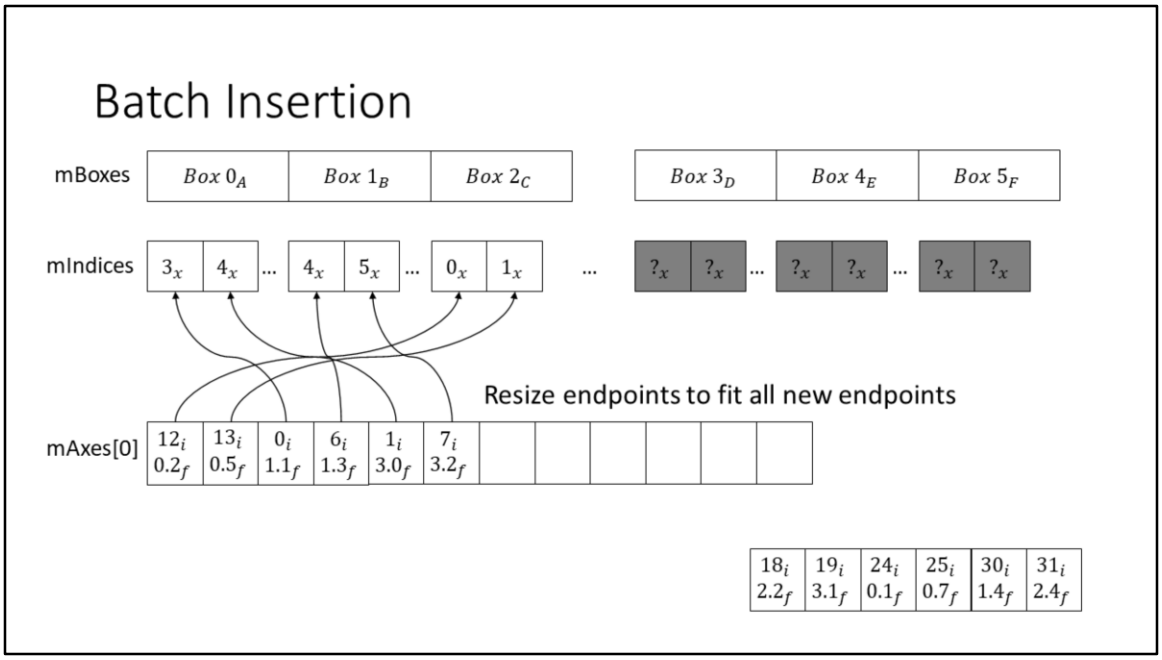
1. Find boxes/indices for each object and resize the endpoints
2. Reverse sort the new endpoints
3. Move the largest of the old and new endpoint to the end of the endpoints

A second pass is performed to find pairs (sweep pass)

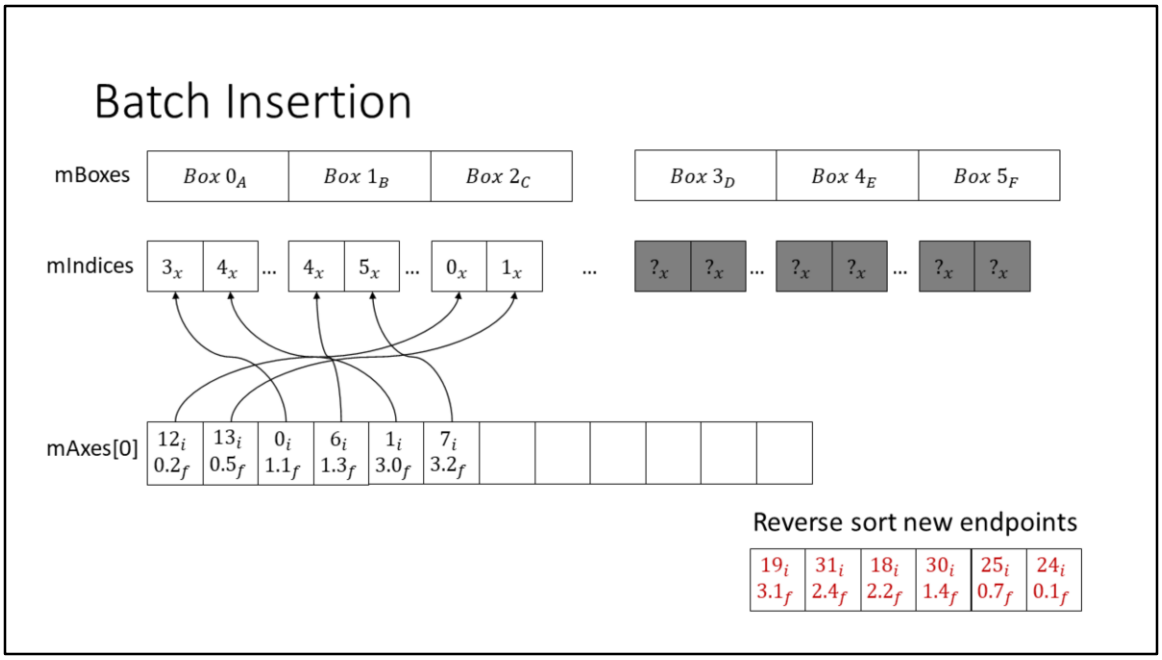
Batch insertion is more straightforward than removal and can simply be summarized in the 3 steps above (pictures to follow). After sorting the endpoints pairs have to be determined in a second pass. This algorithm means we do at worse 2 passes over the array no matter how many objects there are.



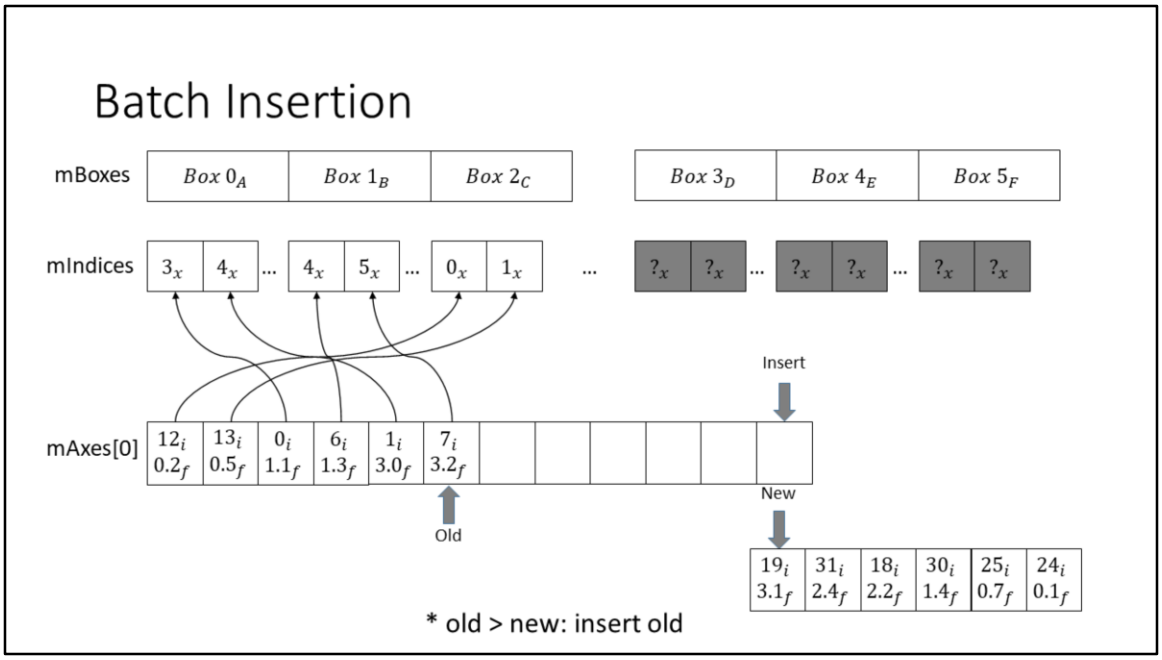
First we find boxes for our new endpoints. At this point the indices for these new boxes don't yet mean anything.



The first step is to allocate enough space for the new endpoints

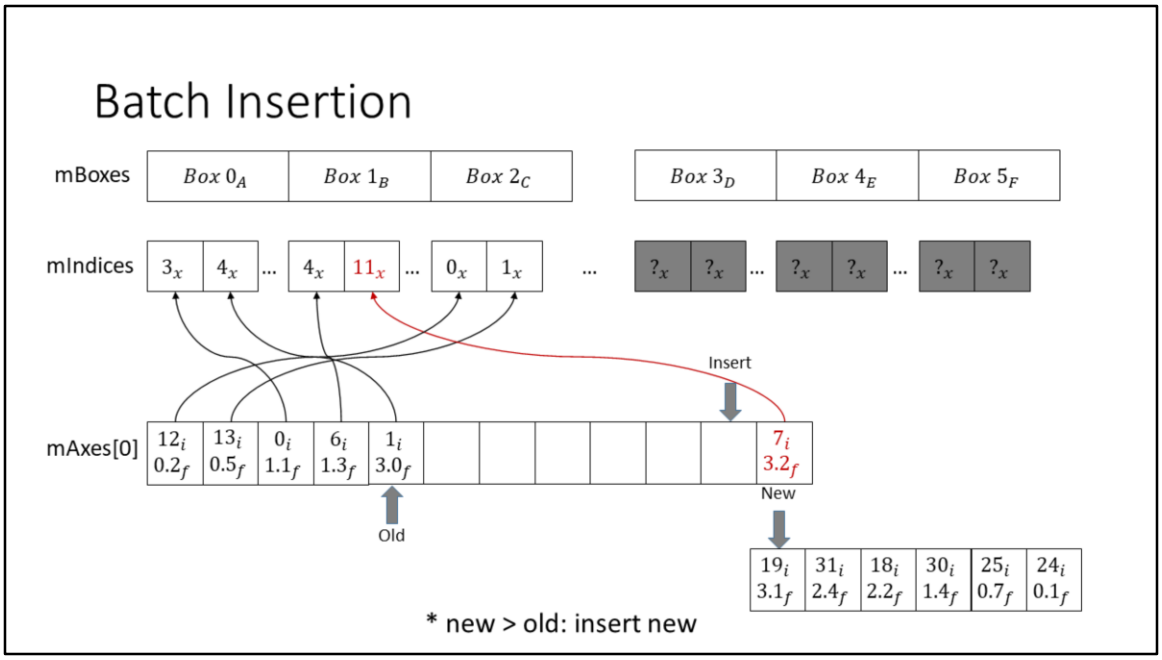


Then we reverse sort the new endpoints. Note that no pair management matters here and we can use our favorite sorting algorithm



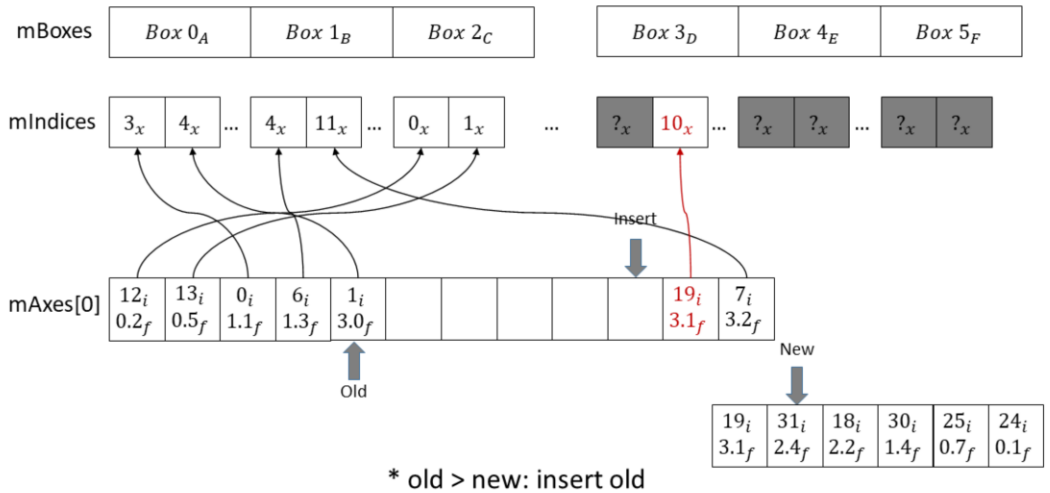
Now we start at the end of the array and choose to move the largest between the end of the old array and the largest of the new array. If we choose from the old array then we decrement our index. If we choose from the new array then we increment. Either way we decrement the insertion point.

In this case the old position is larger than the new one so we move the old.



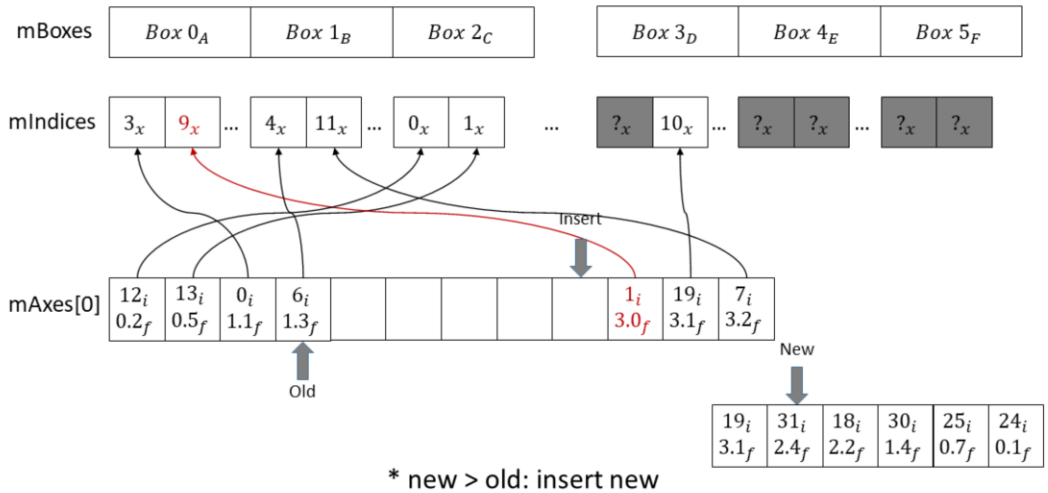
This will continue until we exhaust the new endpoints.

Batch Insertion



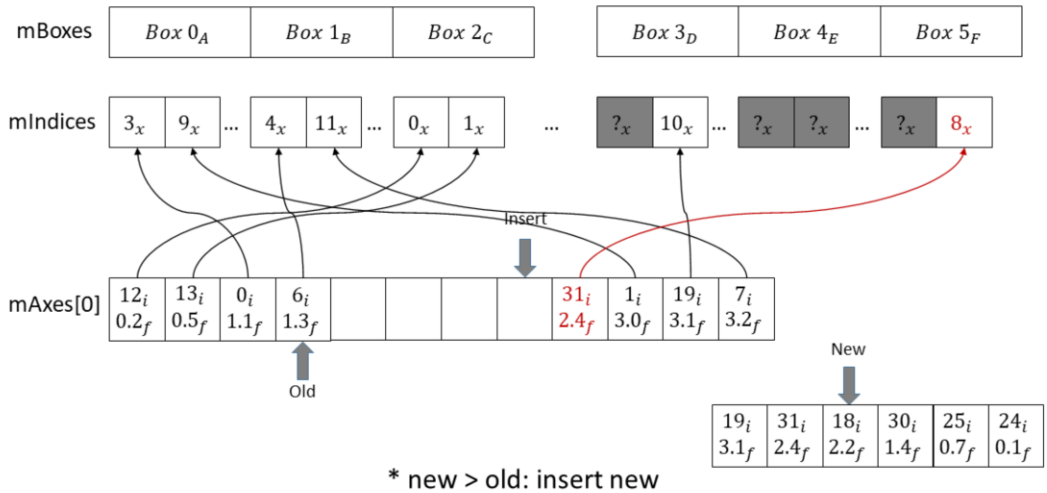
* old > new: insert old

Batch Insertion

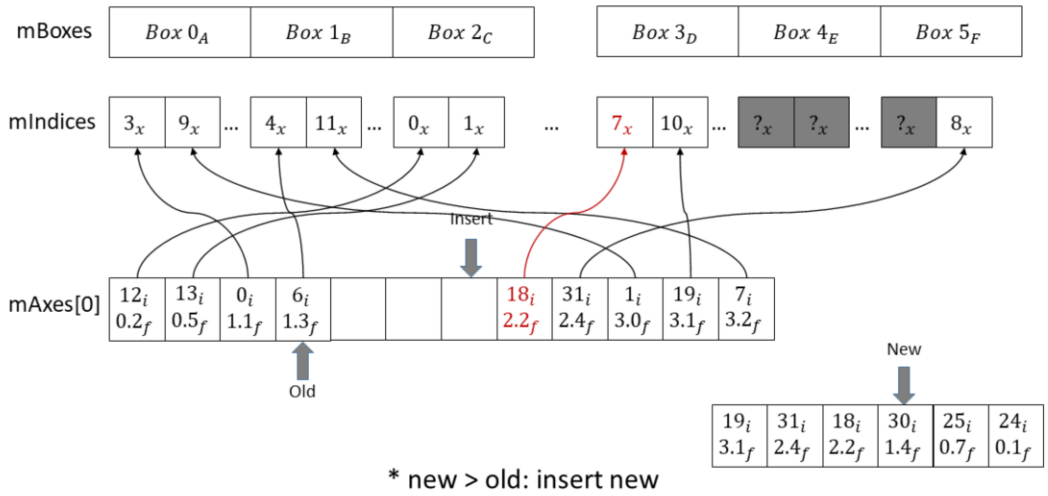


* new > old: insert new

Batch Insertion

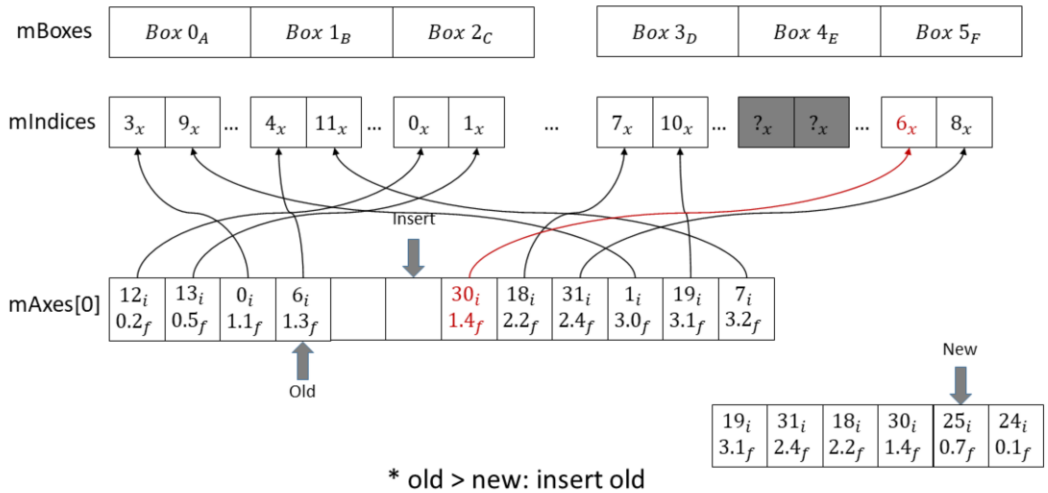


Batch Insertion



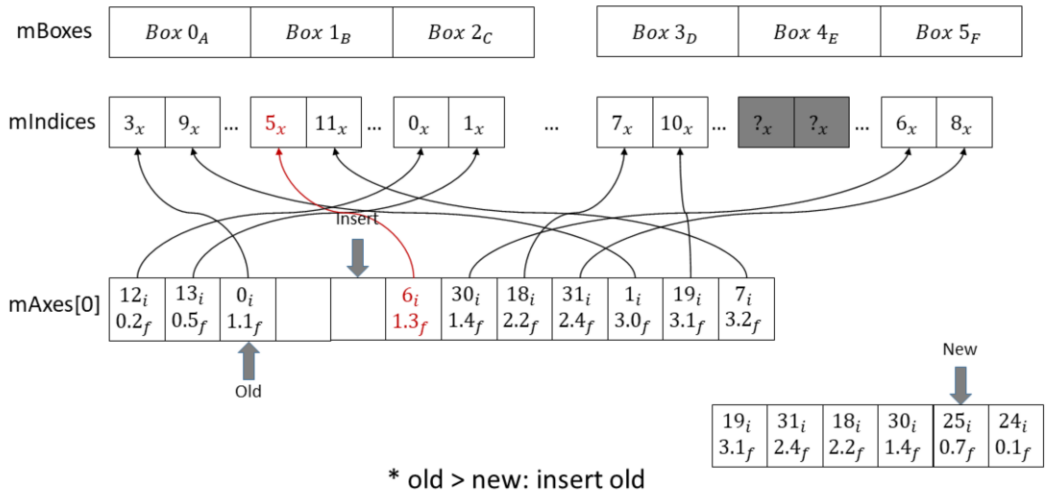
* new > old: insert new

Batch Insertion

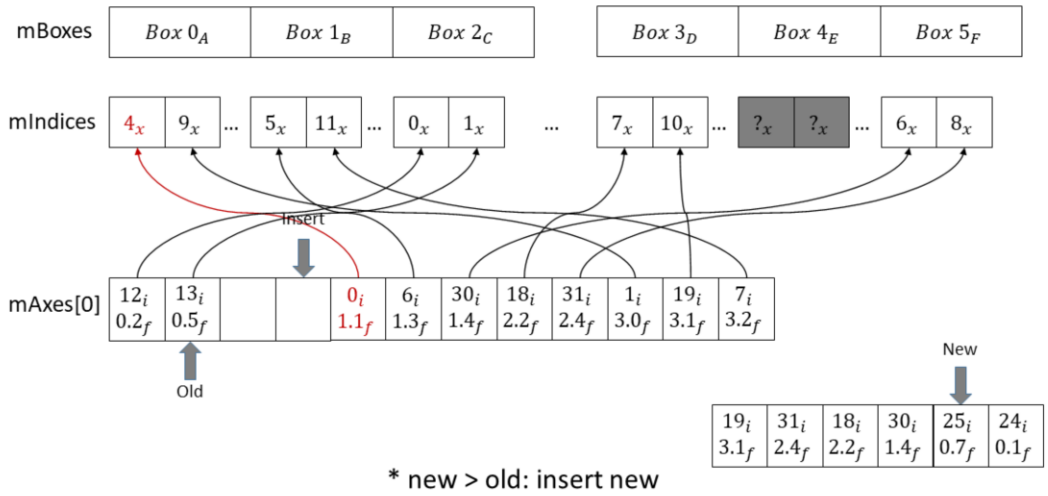


* old > new: insert old

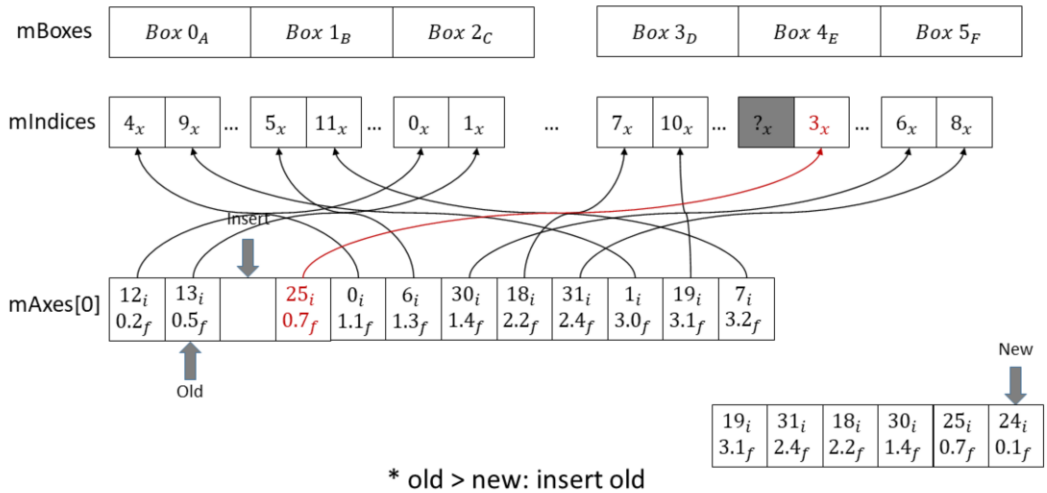
Batch Insertion



Batch Insertion

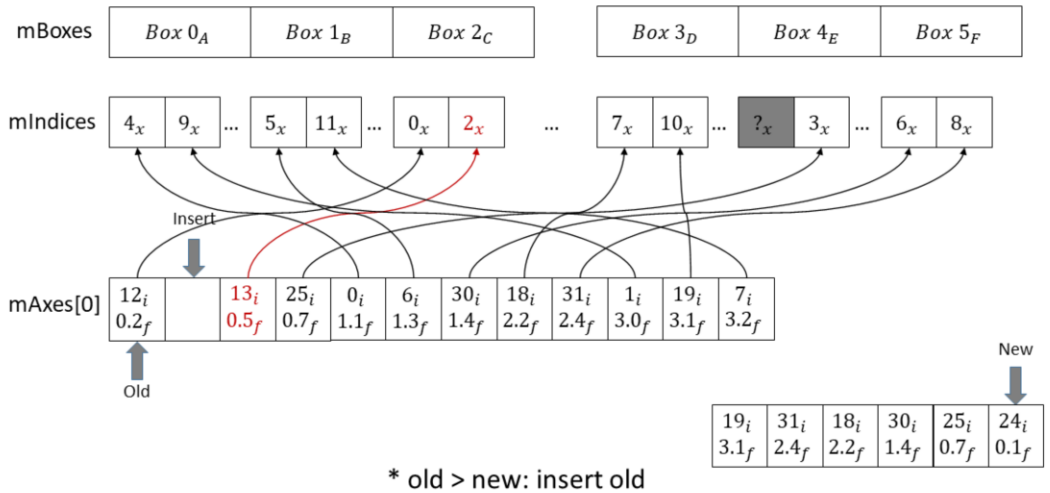


Batch Insertion

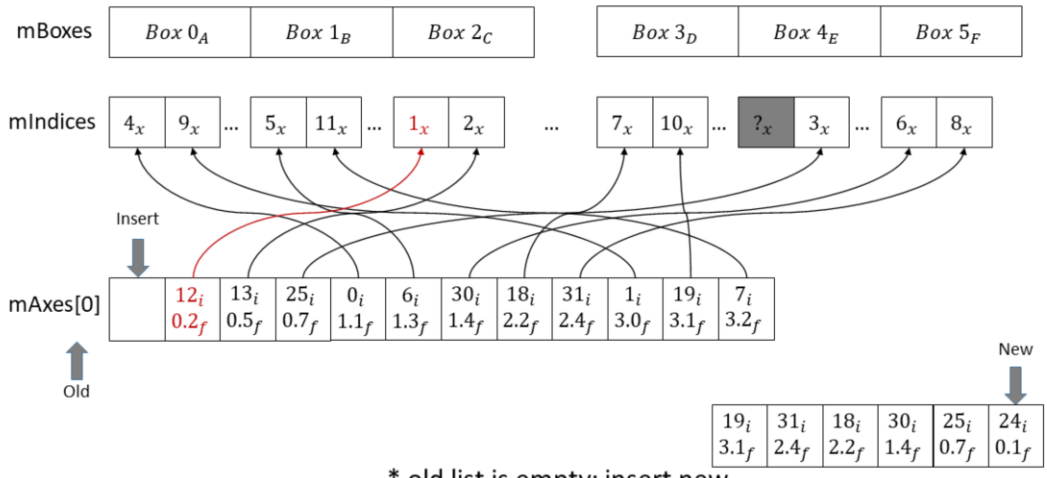


* old > new: insert old

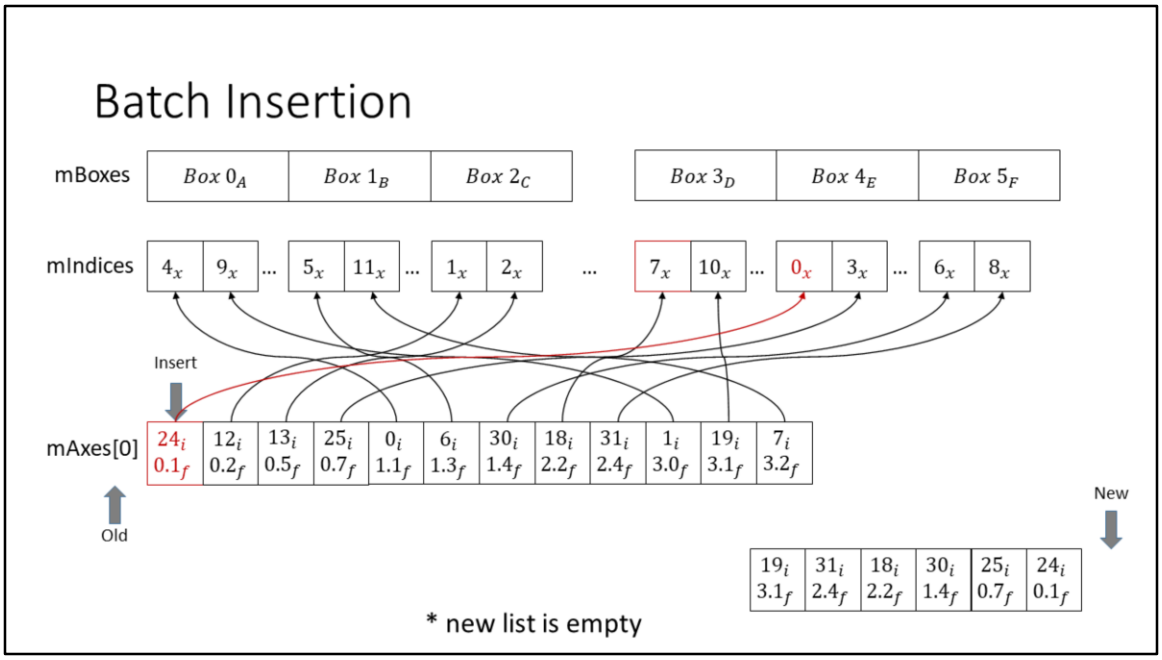
Batch Insertion



Batch Insertion



* old list is empty: insert new



We finally stop when the new list is empty.

Batch Insertion - Pairs

Now endpoints are sorted, but we still need pairs

Perform sweep algorithm (backwards)

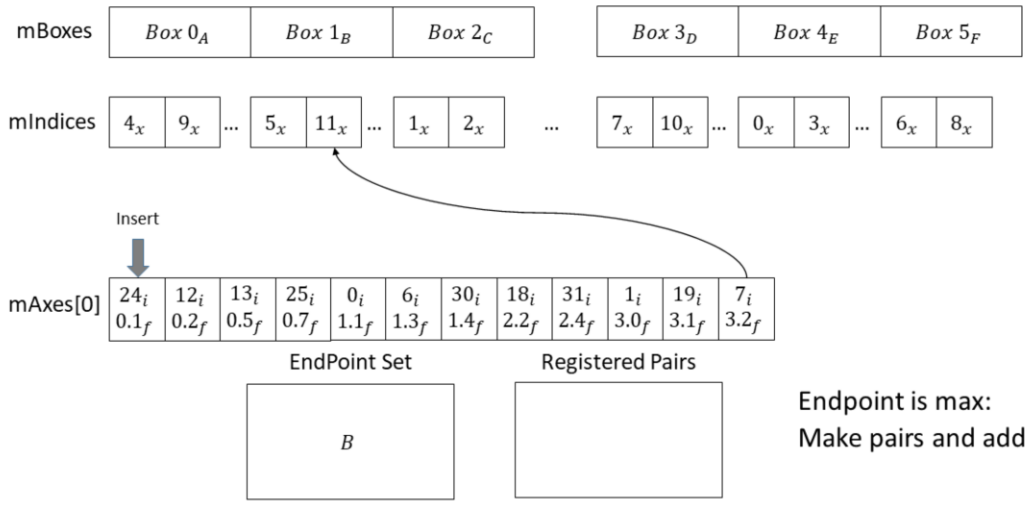
```
if(mEndpoints[i].IsMax())
{
  // Create pairs with everything in our map
  // Add this object to the map
}
else
  // Remove from the map
```

Start at end and continue until last insertion point

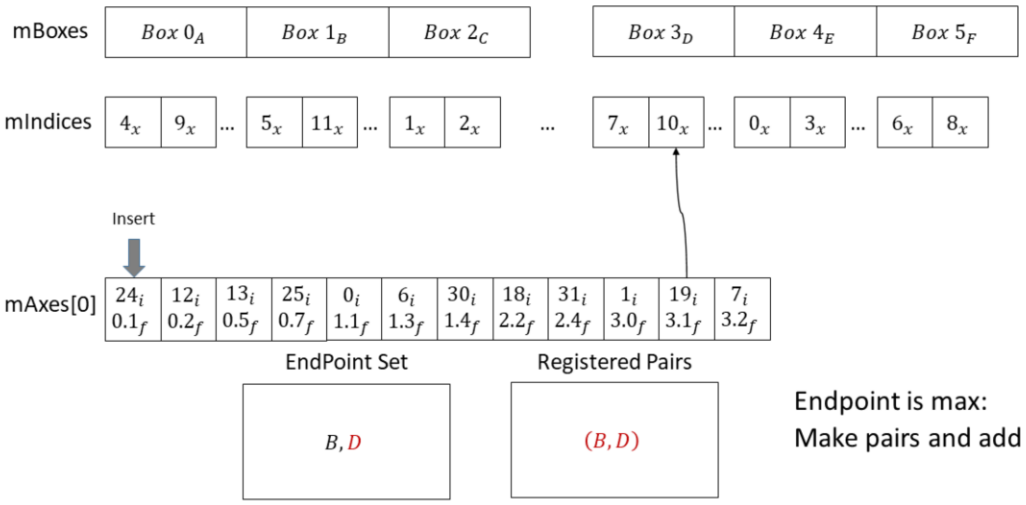
*Only 1 axis needs to be swept

Now we just need to create pairs. We've actually already covered this algorithm, it's just the simple linear sweep to find pairs but this time in reverse. We have to start at the end and continue until we reach the last insertion point (since we didn't necessarily stop at the beginning of the array). Also only 1 axis needs to be swept over, although the remaining axis values need to be checked before inserting a pair

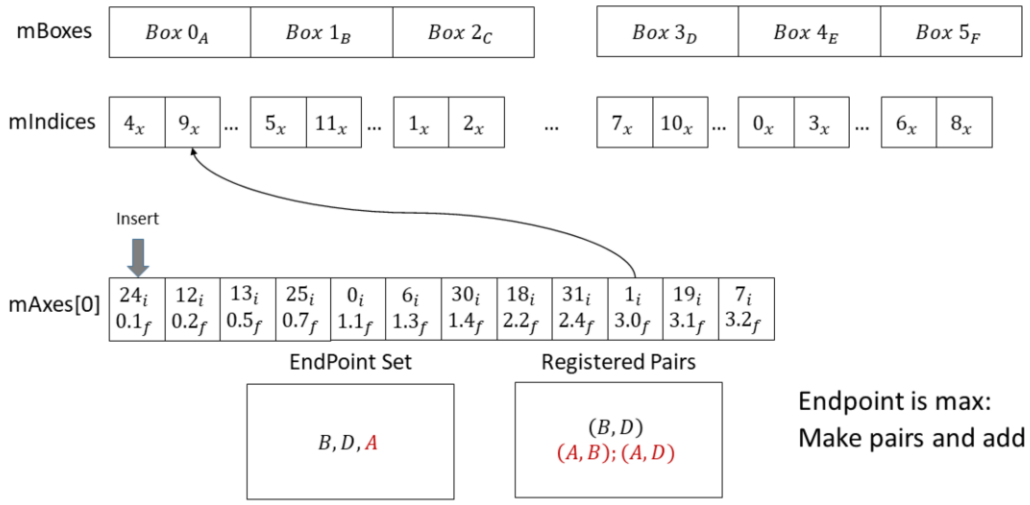
Batch Insertion



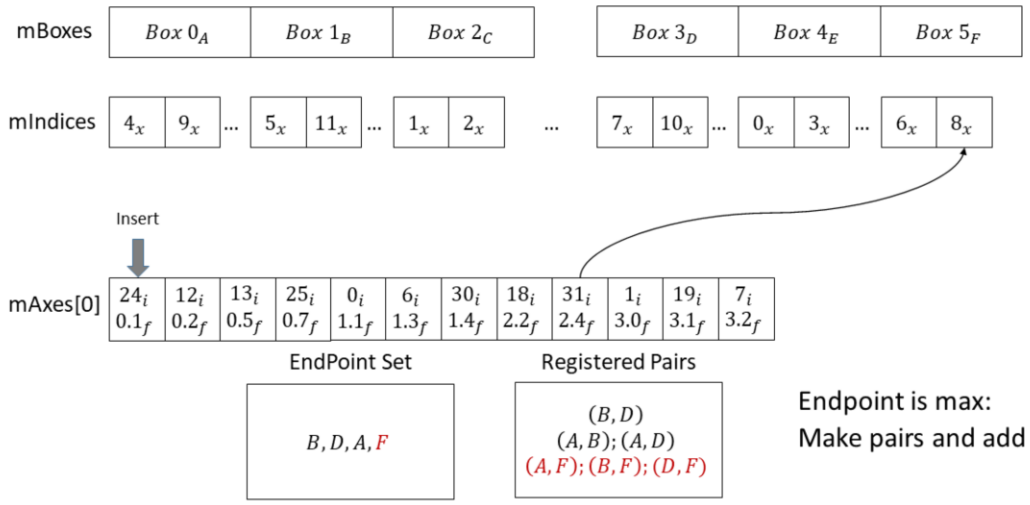
Batch Insertion



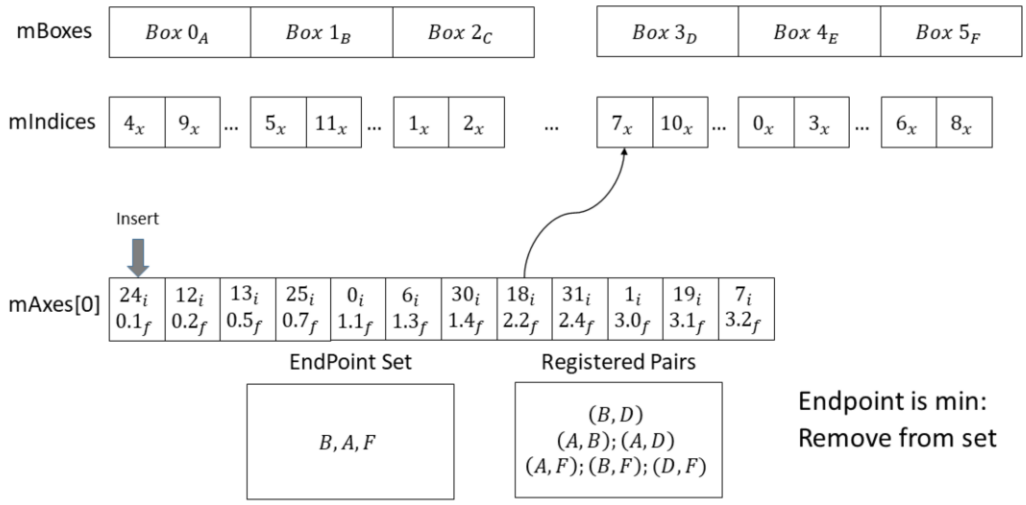
Batch Insertion



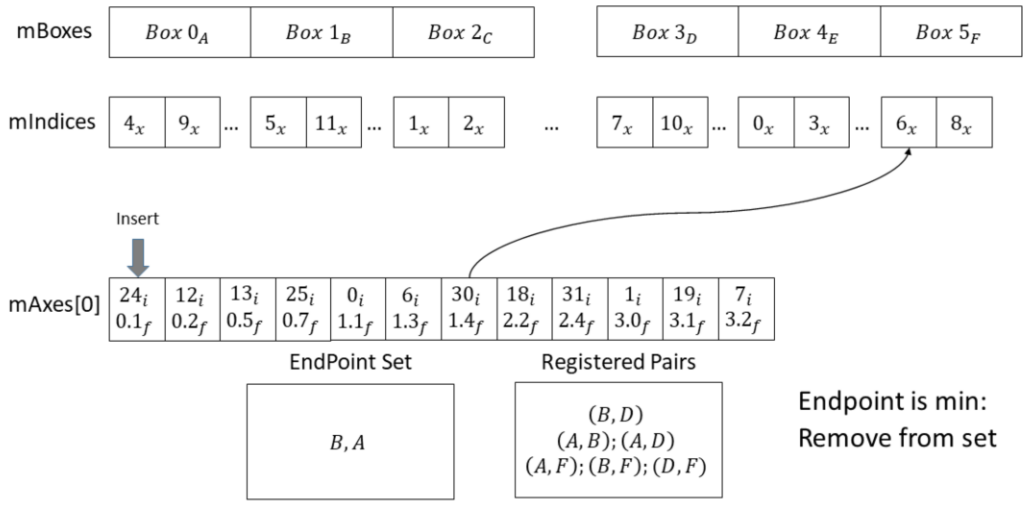
Batch Insertion



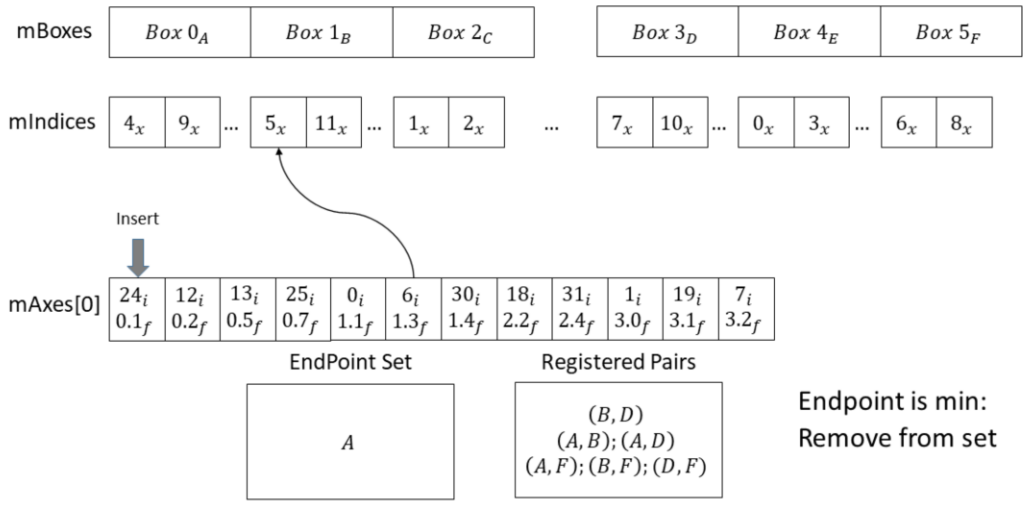
Batch Insertion



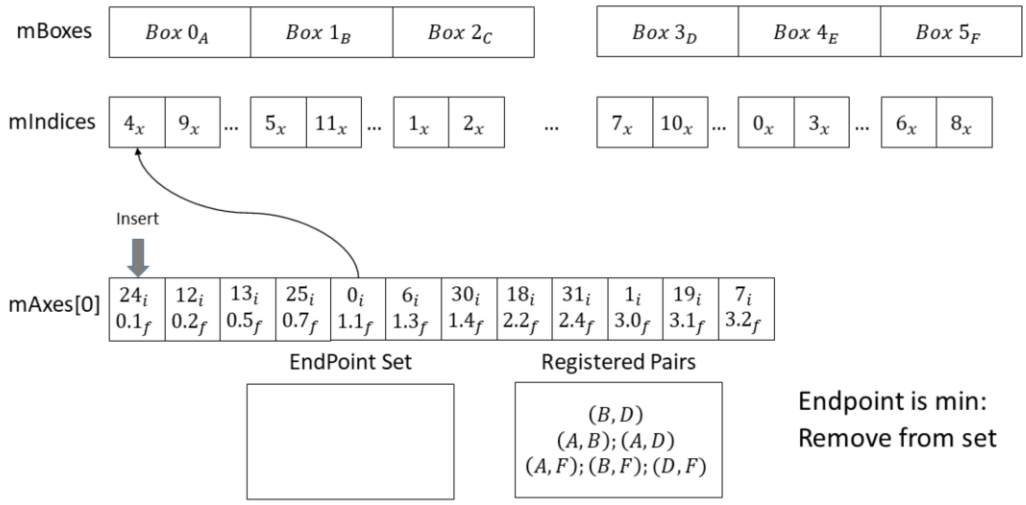
Batch Insertion



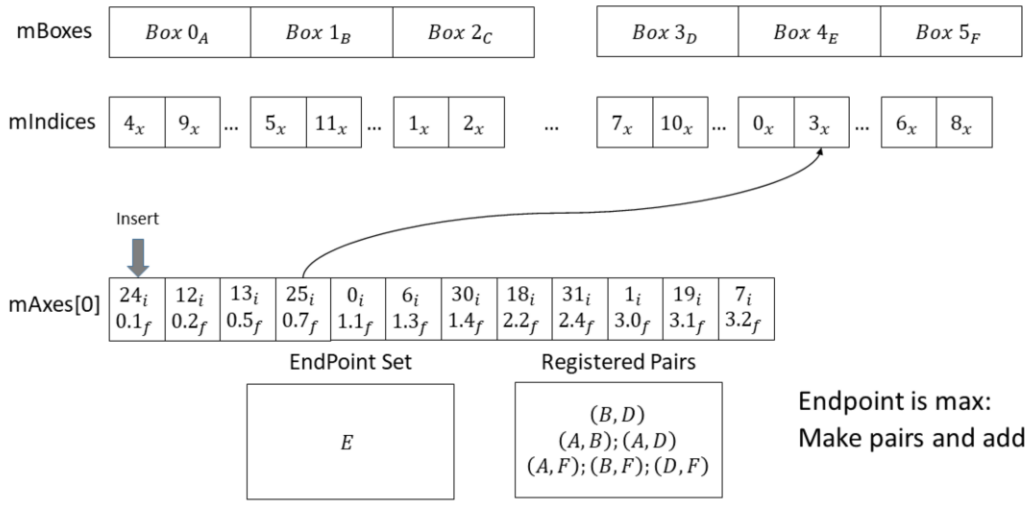
Batch Insertion



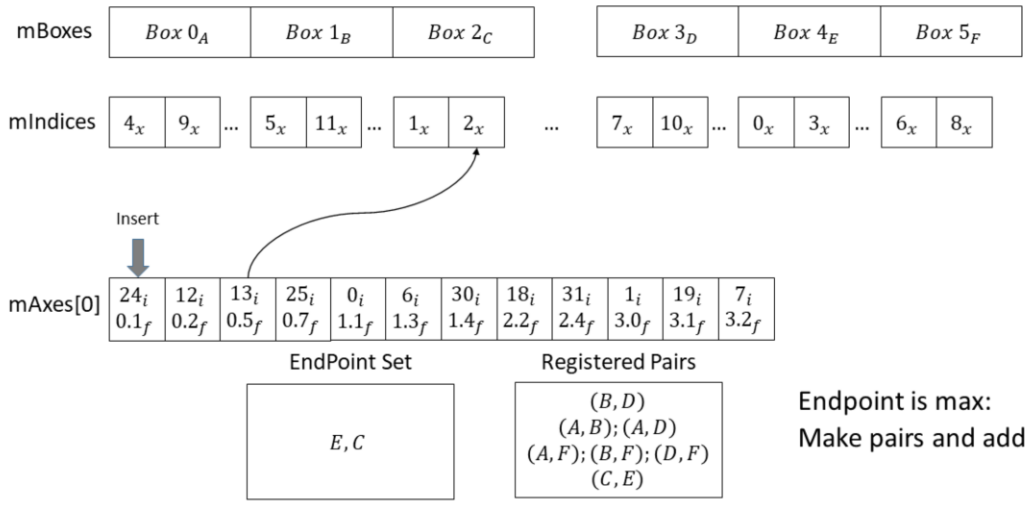
Batch Insertion



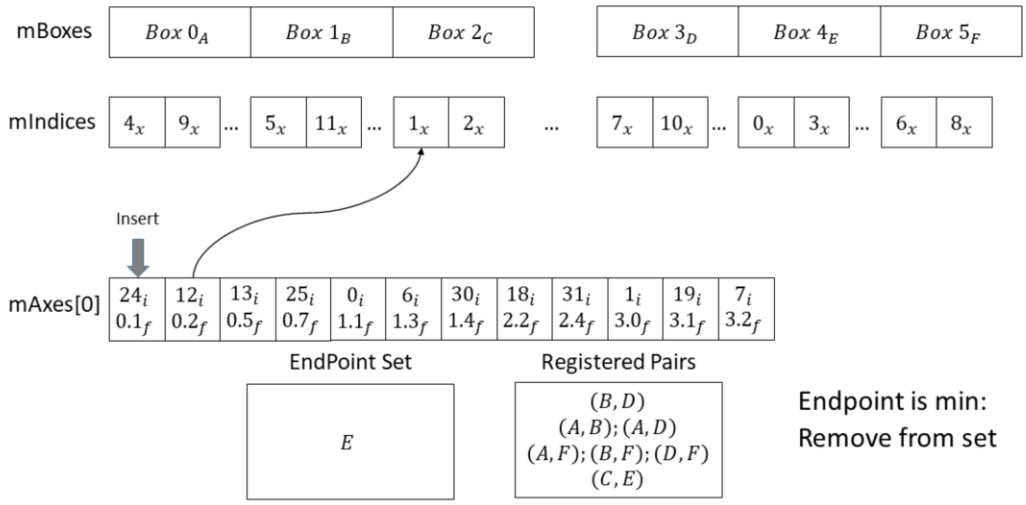
Batch Insertion



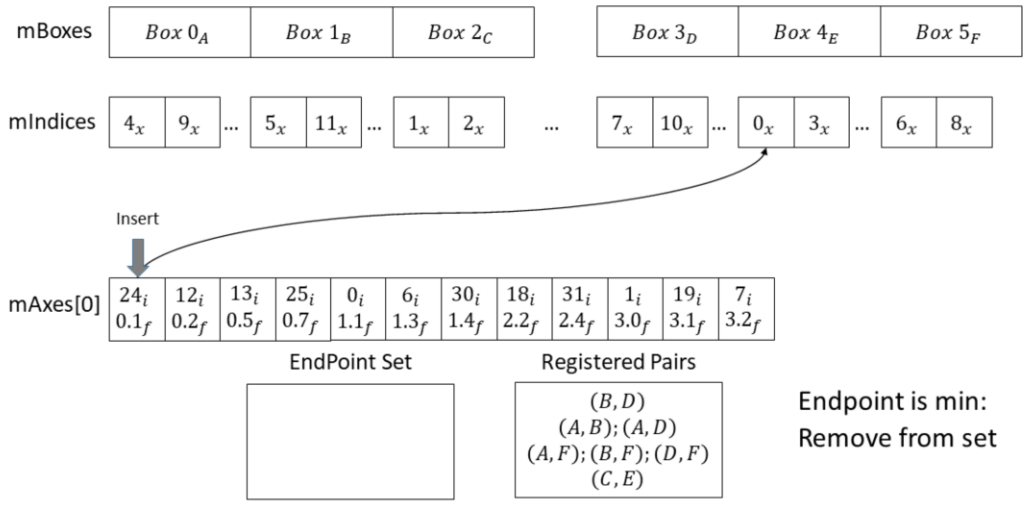
Batch Insertion



Batch Insertion



Batch Insertion



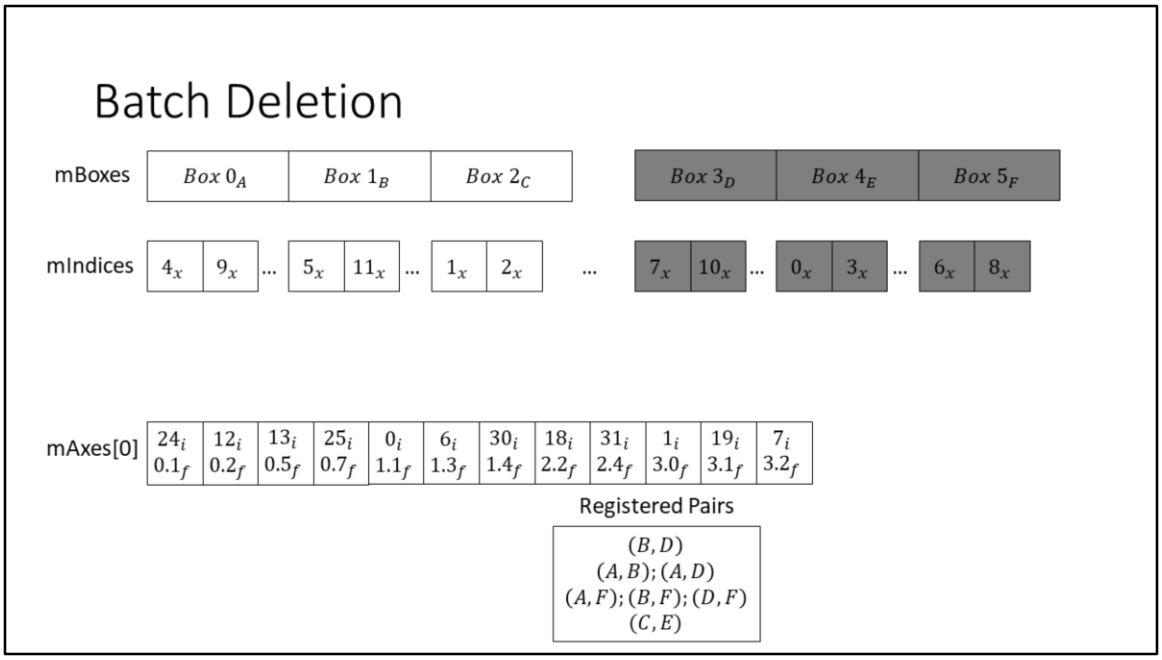
Batch Deletion

Deletion is more complicated

Can be broken up into 3 large-ish steps:

1. Mark boxes as invalid
2. Remove pairs
3. Remove Endpoints

Deletion is a more difficult but can be broken up into 3 main steps. Each of these steps is a fairly complicated.



The first step is to mark all boxes that we're going to stop using as invalid.

Batch Deletion – Pair Removal

Bi-parte box pruning:

- Remove overlapping pairs of only objects we're removing (keep old pairs)

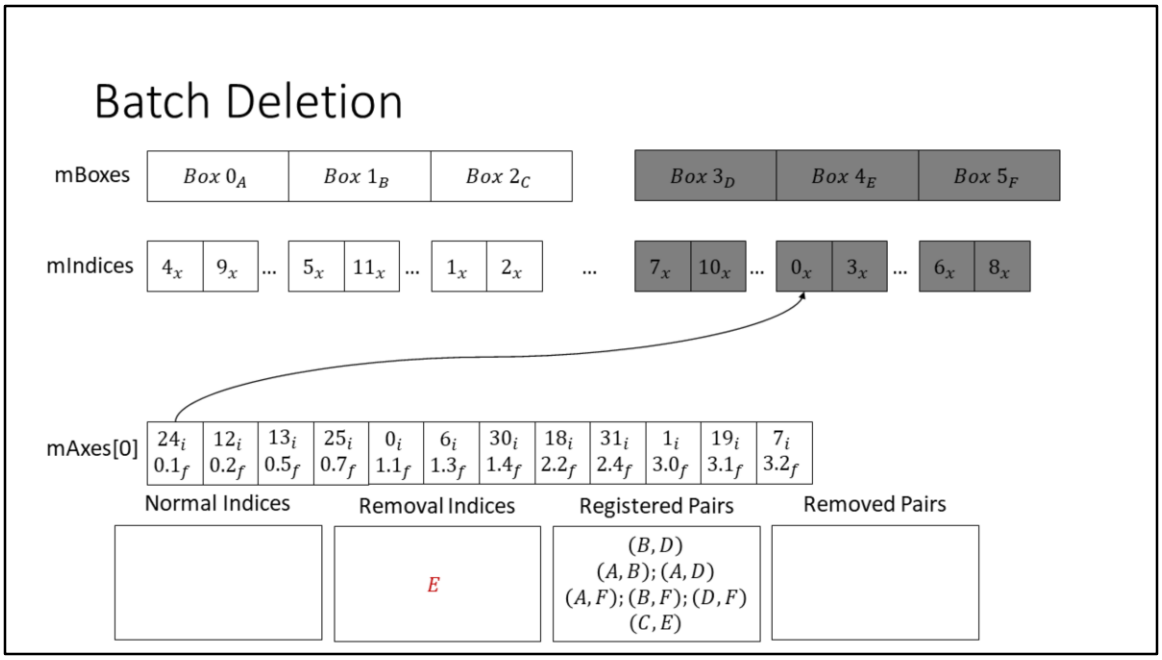
If an endpoint is a Min:

- If being deleted:
  - Remove pairs with both sets and add to removal set
- Else
  - Remove pairs only with removal set, add to old set

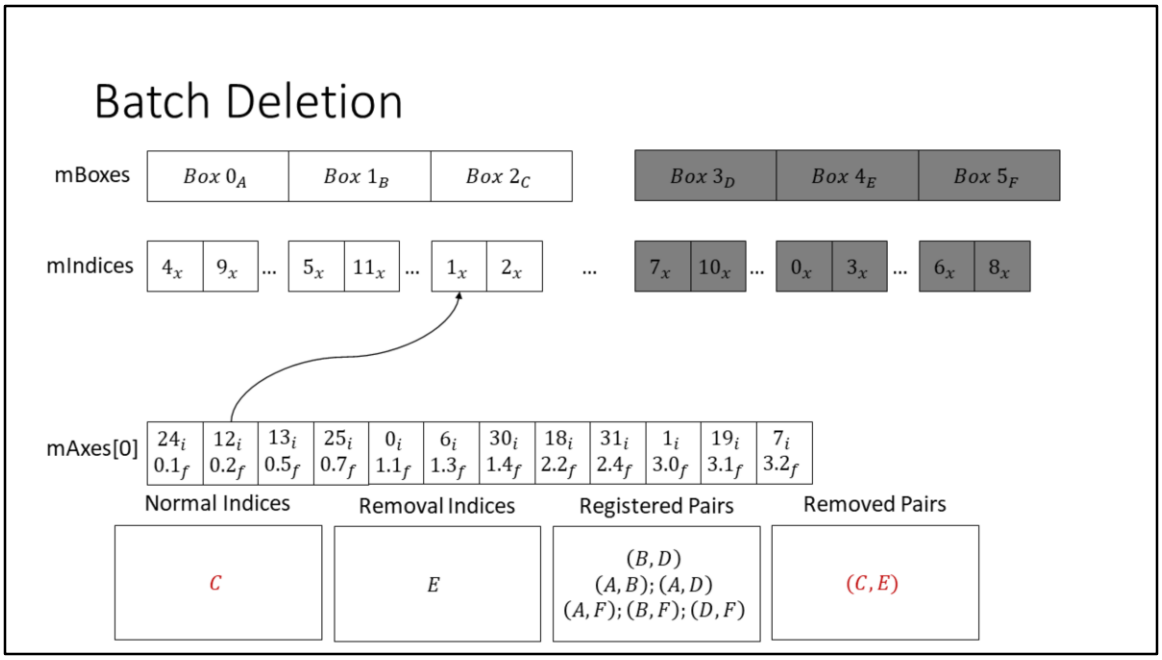
Else if max:

- Remove from the correct set

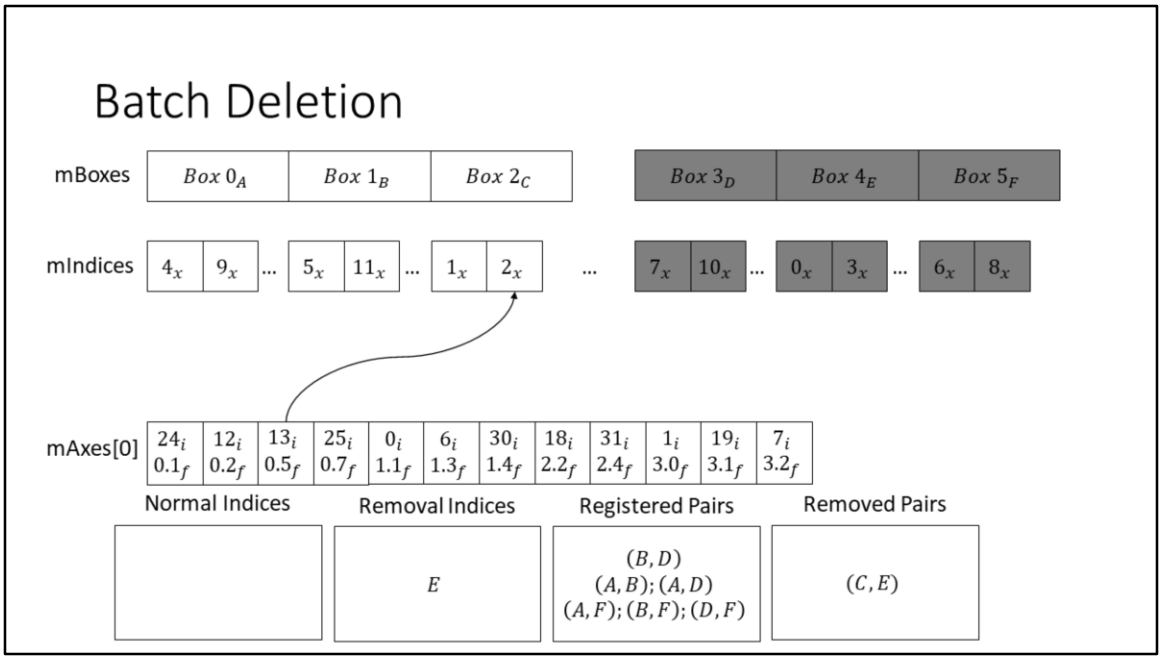
Removing pairs is a lot more complicated. This is known as a bi-parte box pruning algorithm as we only want to remove some pairs and not others. Any pairs with boxes that are being deleted we want to get rid of, but pairs of objects not being deleted need to stay around. We achieve this by keeping two endpoints sets, one of endpoints being removed and the other of normal endpoints.



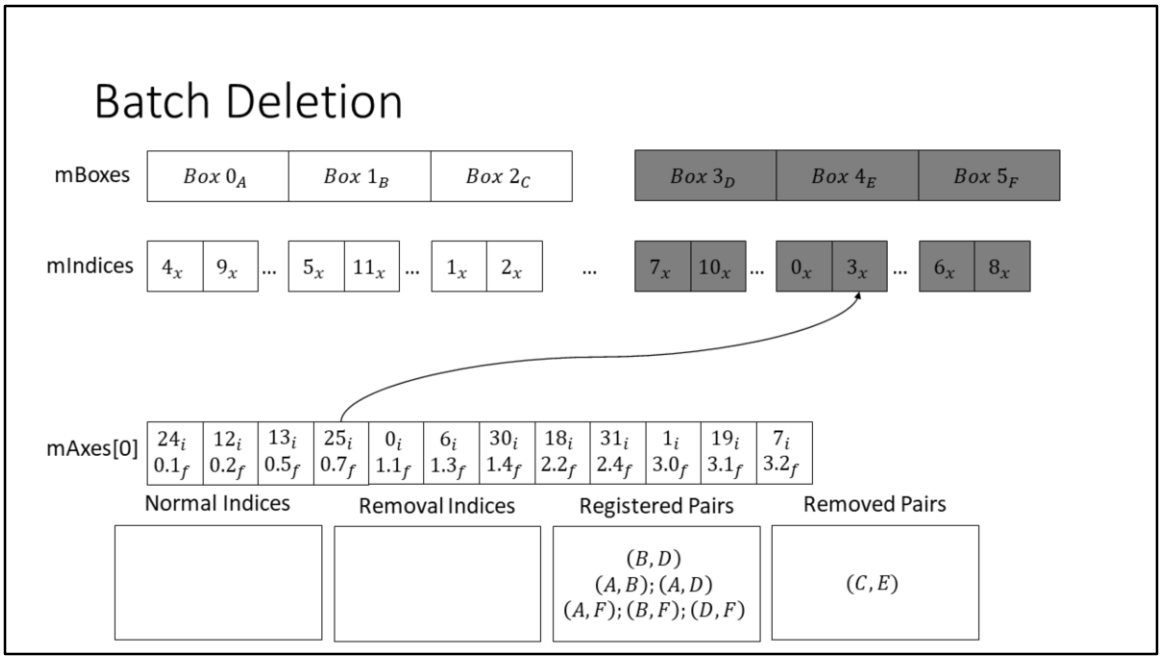
The first endpoint is a min we're removing, we remove pairs with everything in the normal and removal indices and then add to the removal indices. In this case we just add to removal indices.



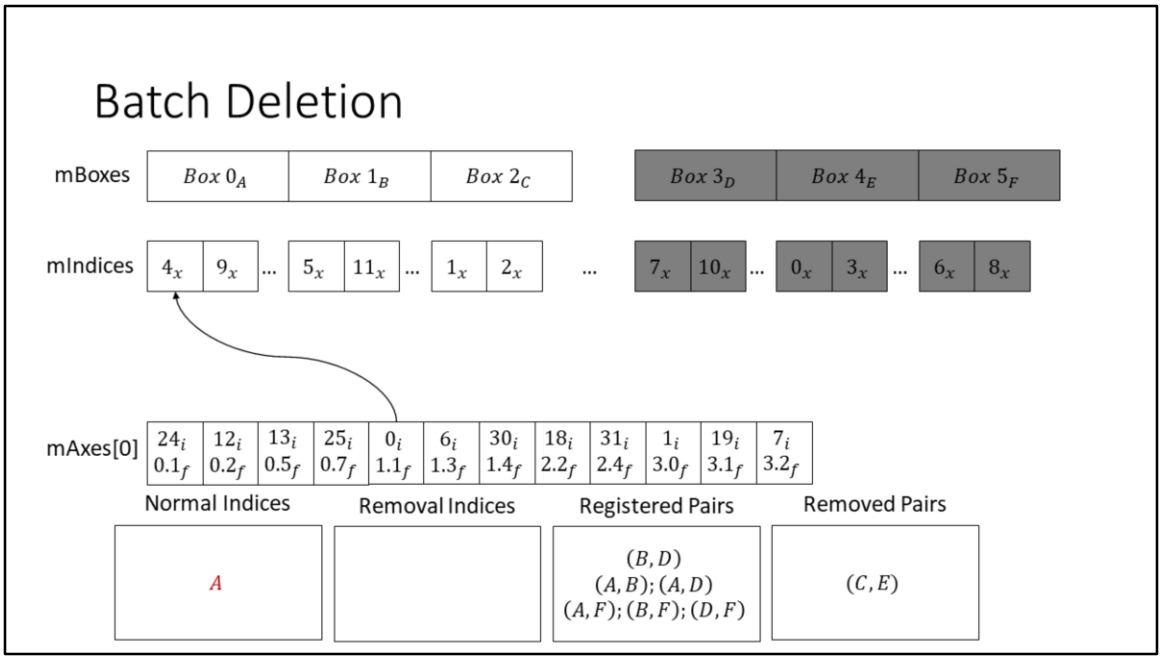
We now have a min we aren't removing. We remove pairs with everything in the removal indices (not the normal ones) and then add to the normal indices.



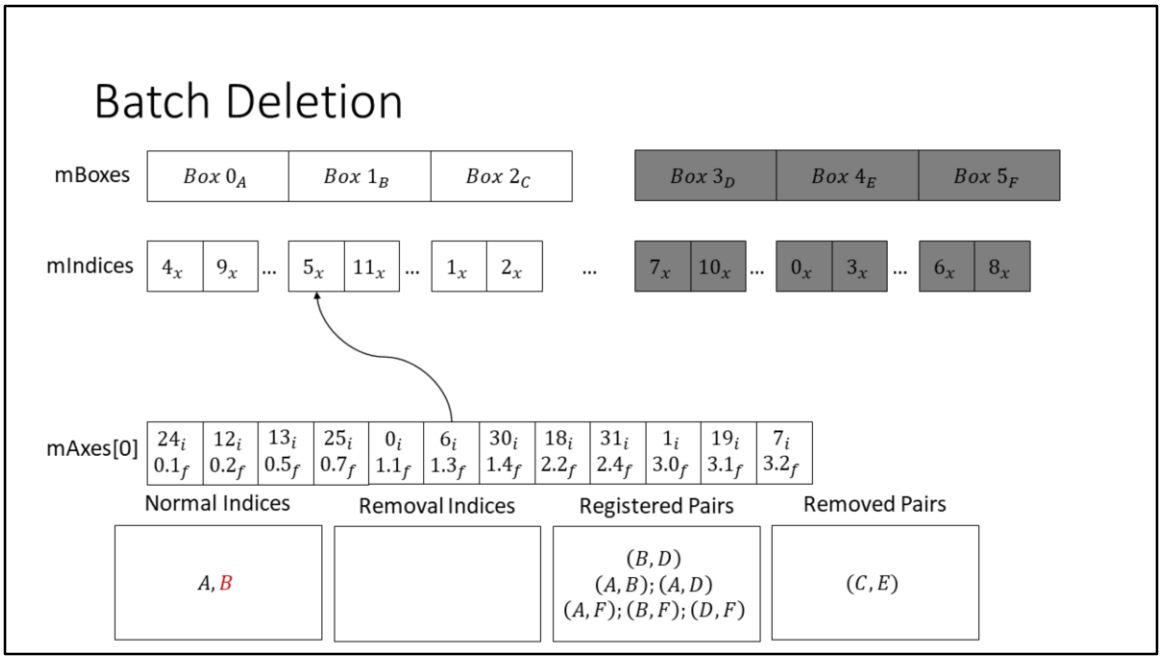
We just remove max endpoints from whatever set they were in.



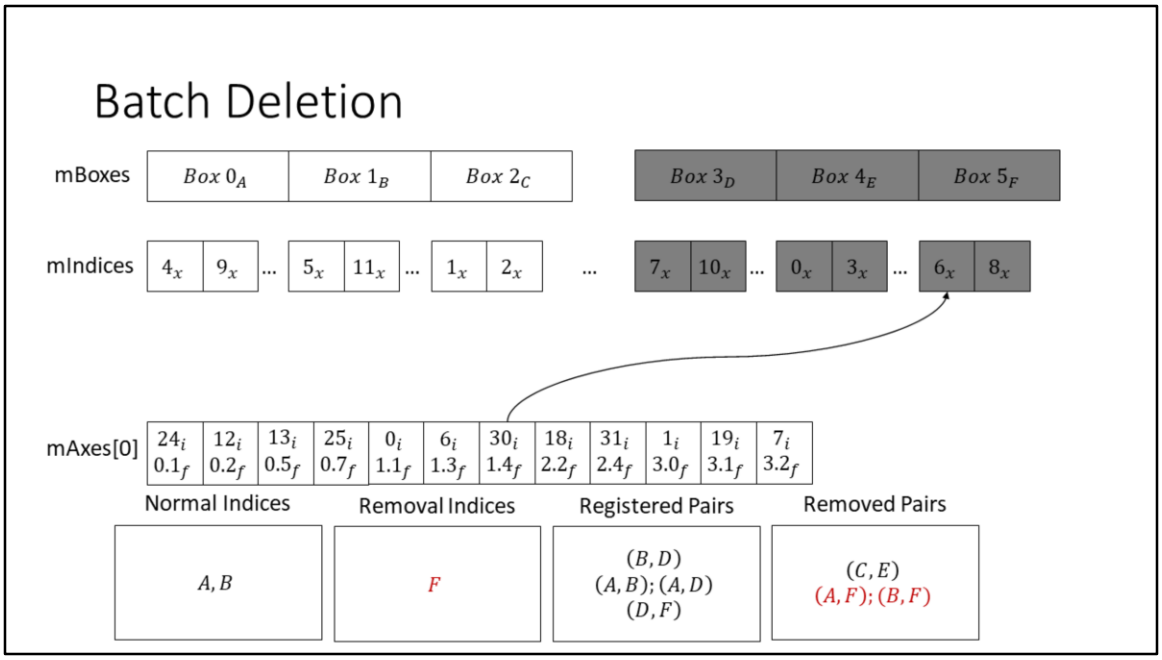
We just remove max endpoints from whatever set they were in.



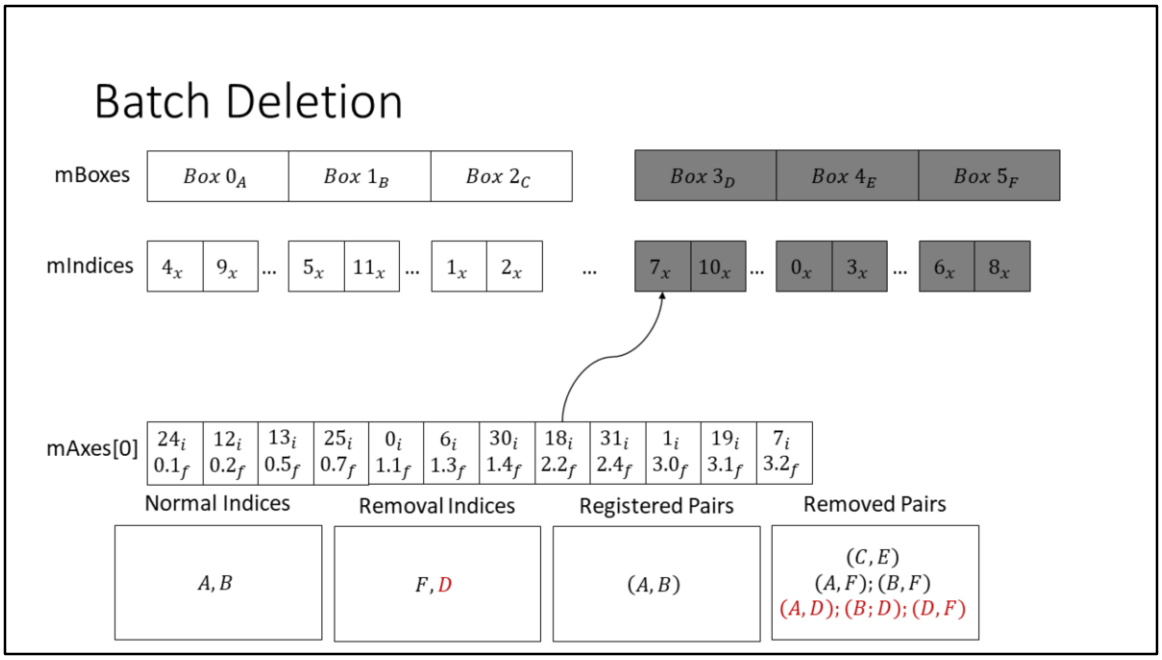
Min that isn't being removed: remove pairs with everything in the removal indices (not the normal ones) and then add to the normal indices.



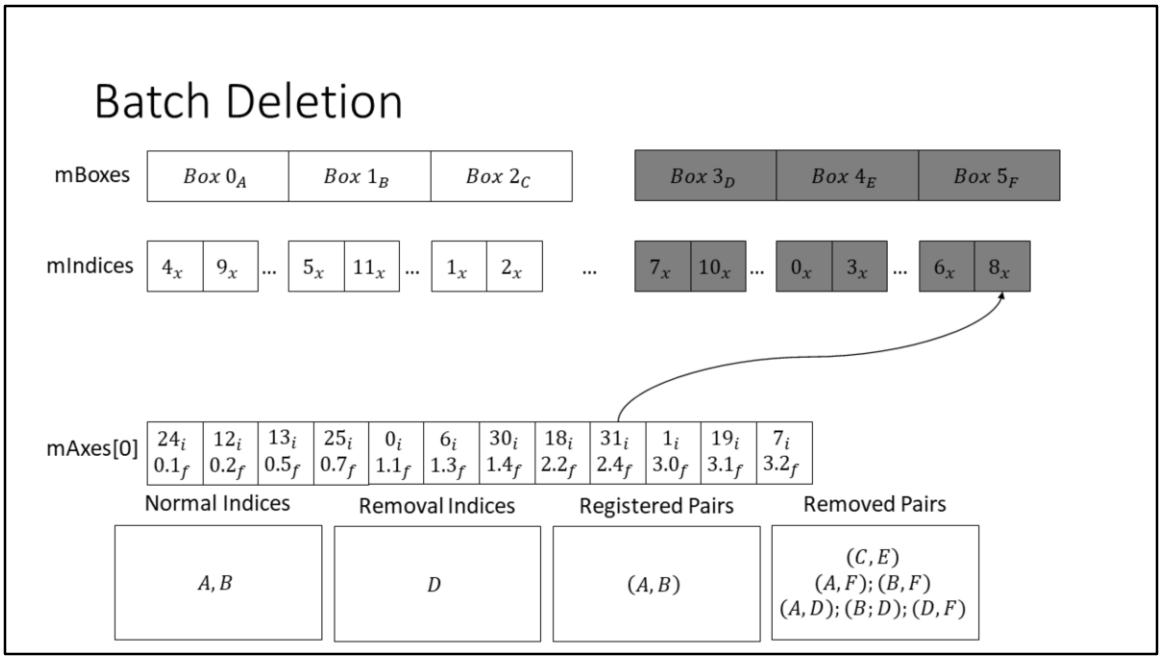
Min that isn't being removed: remove pairs with everything in the removal indices (not the normal ones) and then add to the normal indices.



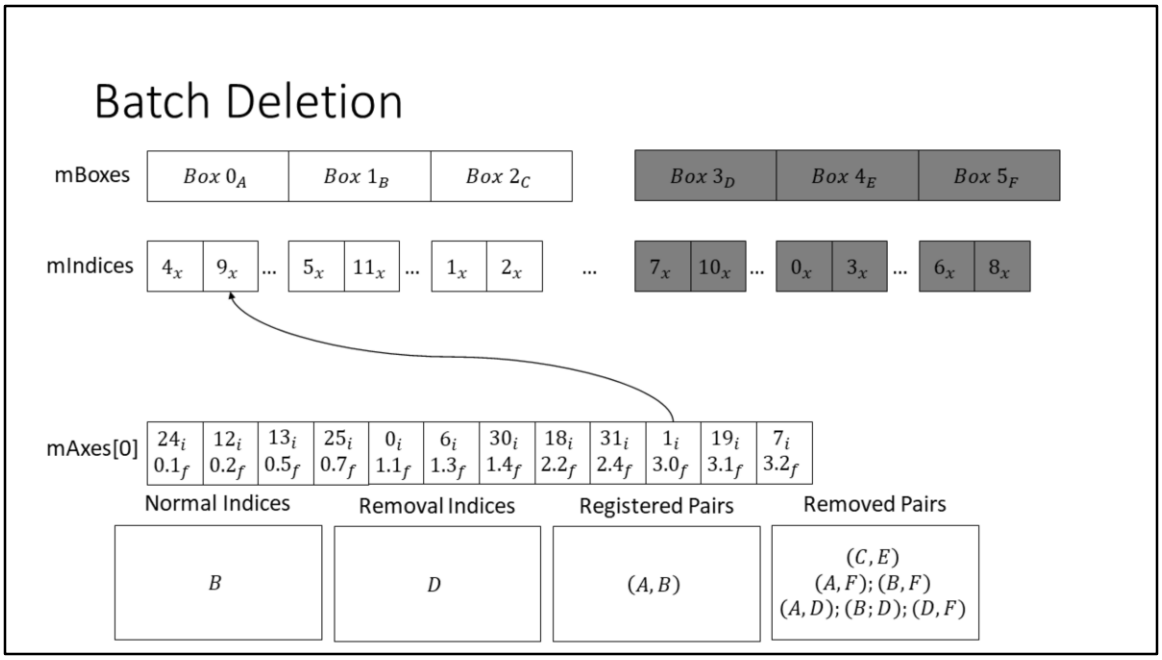
Min that is being removed: remove pairs with everything in the both sets and then add to the removal indices.



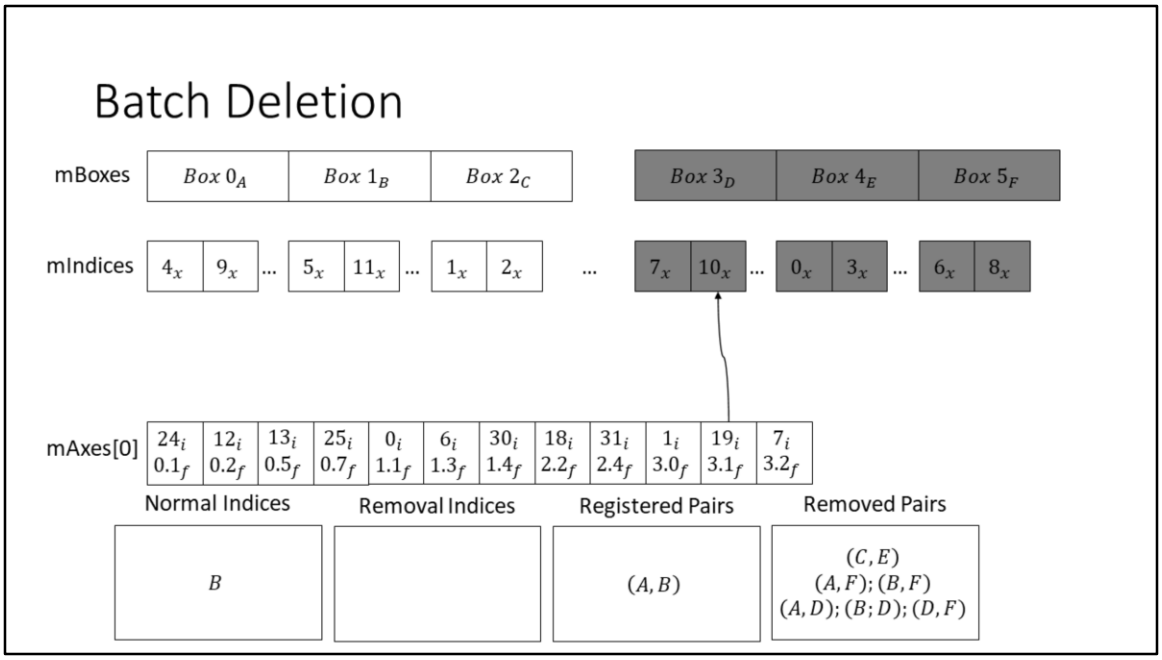
Min that is being removed: remove pairs with everything in the both sets and then add to the removal indices.



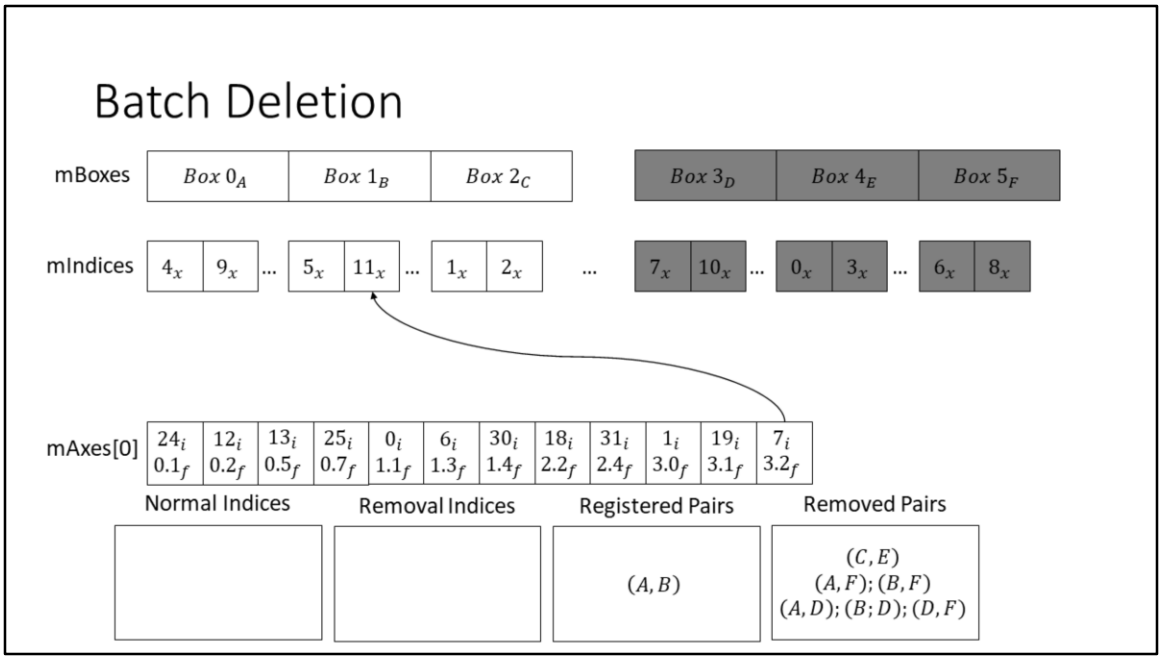
Max endpoint: remove from whatever set it was in.



Max endpoint: remove from whatever set it was in.



Max endpoint: remove from whatever set it was in.



Max endpoint: remove from whatever set it was in.

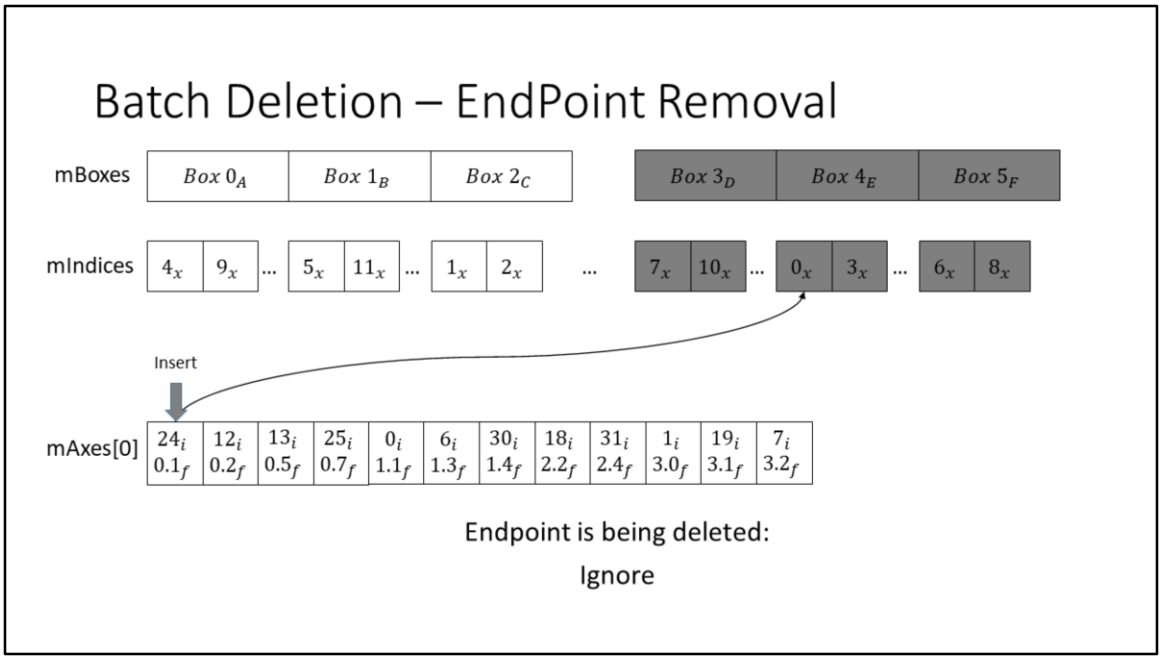
Batch Deletion – EndPoint Removal

Loop from beginning to end of an axis

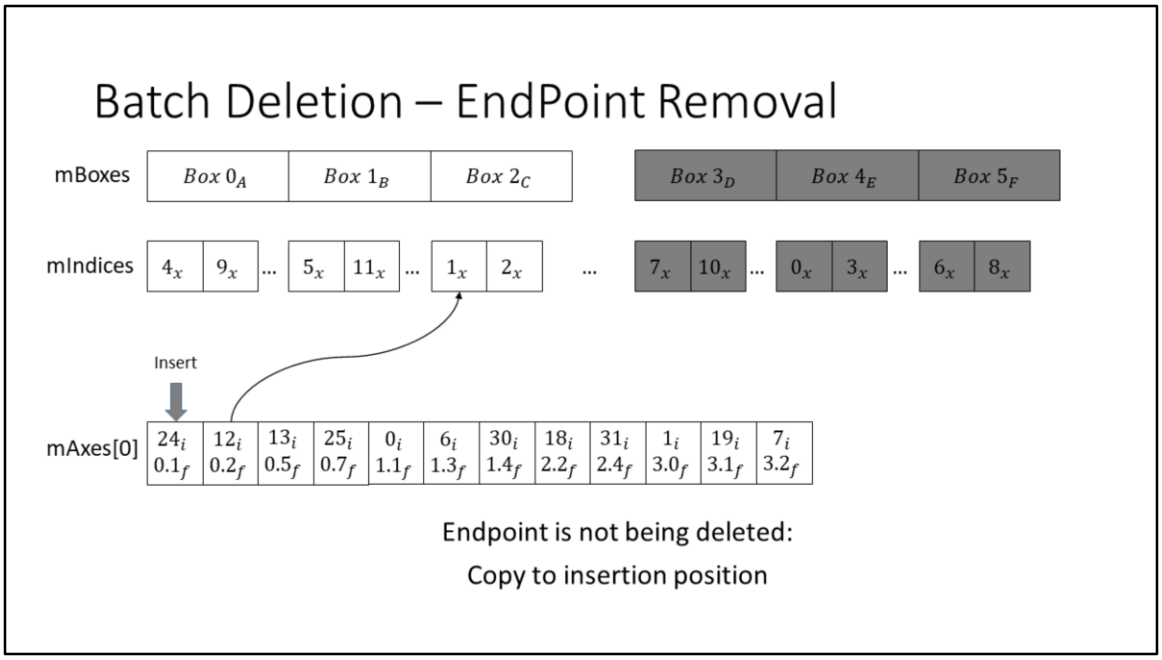
Keep track of separate insertion and test positions:

- If an endpoint is being deleted, skip it
- Otherwise copy to the insertion position

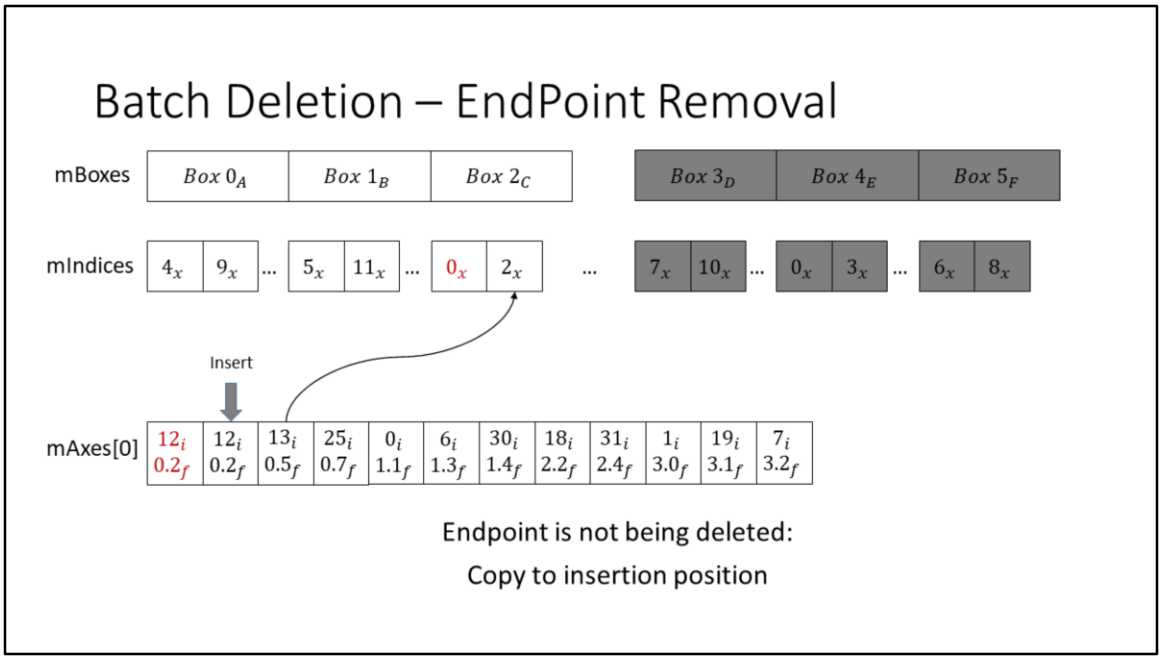
Now we just have to remove the endpoints that are being deleted in one pass. This is achieved by looping from the beginning to end of the endpoint array and collecting all of the old endpoints at the beginning of the array while ignoring the ones being deleted.



The first endpoint is one we are deleting so we want to ignore it. To do this we advance our test position but leave the insertion position alone.

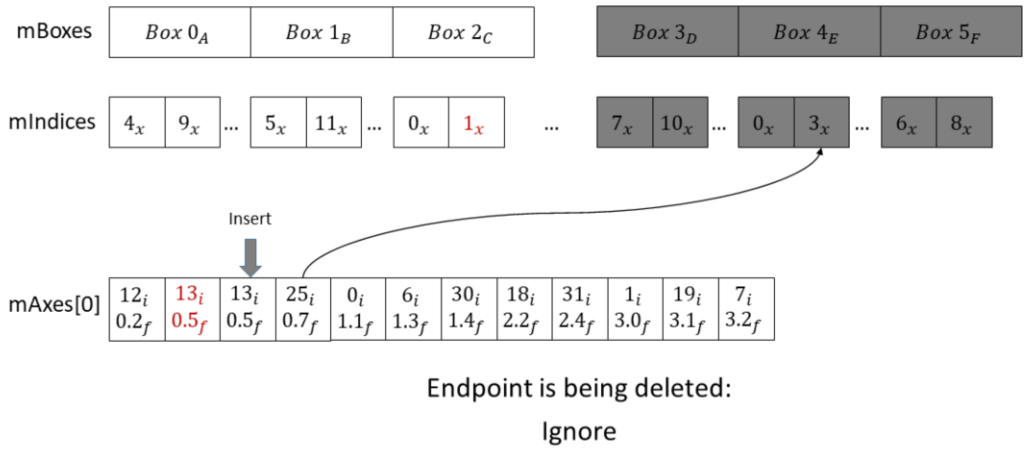


The next endpoint is not being deleted so we need to copy it to the insertion position. After copying we advance both the insertion and test position



This continues until the test position reaches the end of the array

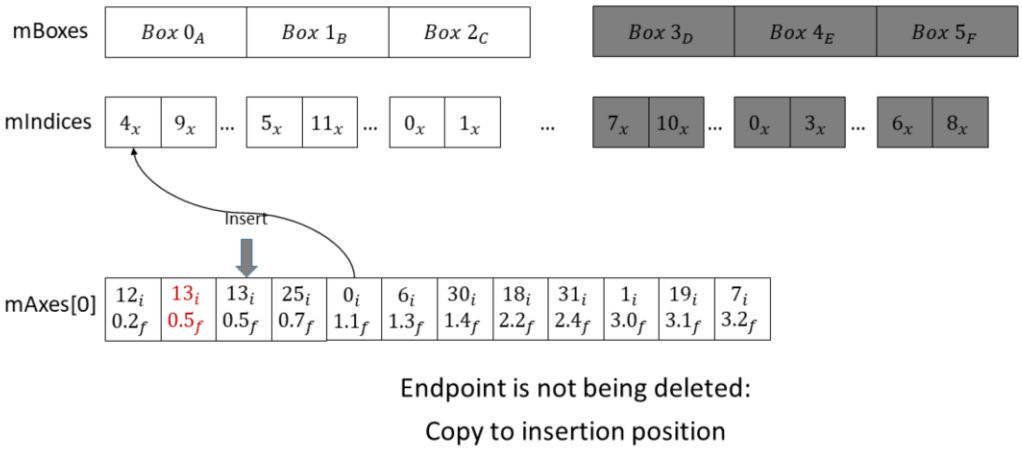
Batch Deletion – EndPoint Removal



Endpoint is being deleted:

Ignore

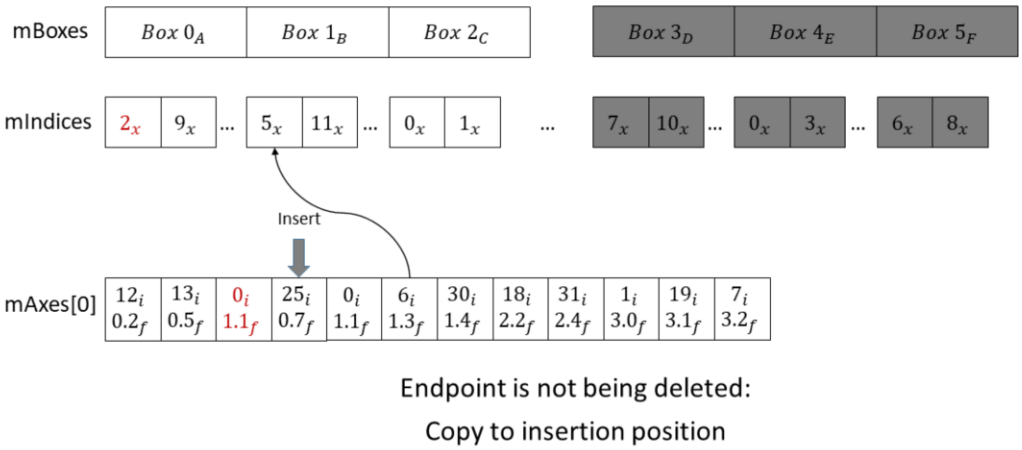
Batch Deletion – EndPoint Removal



Endpoint is not being deleted:

Copy to insertion position

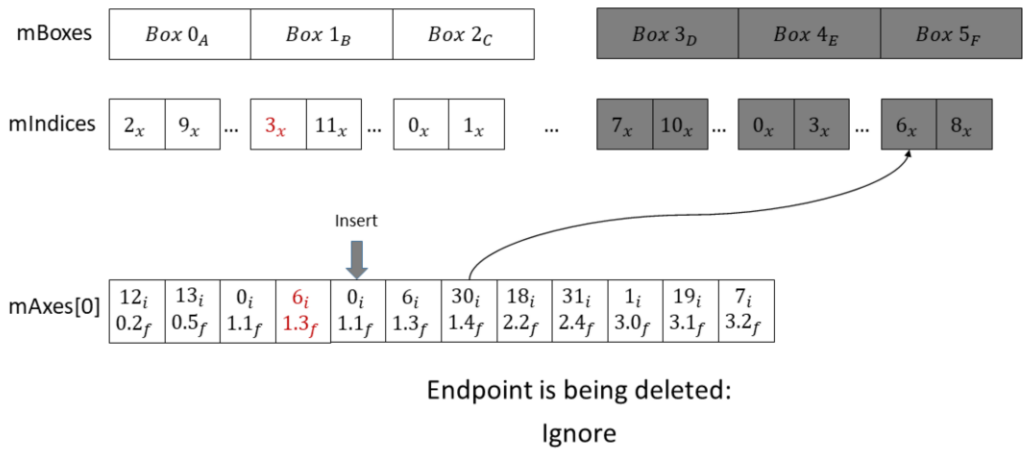
Batch Deletion – EndPoint Removal



Endpoint is not being deleted:

Copy to insertion position

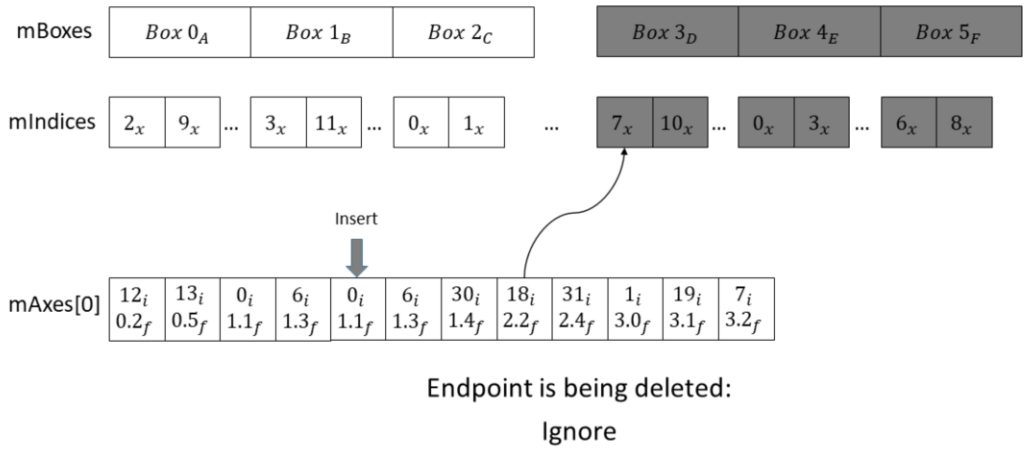
Batch Deletion – EndPoint Removal



Endpoint is being deleted:

Ignore

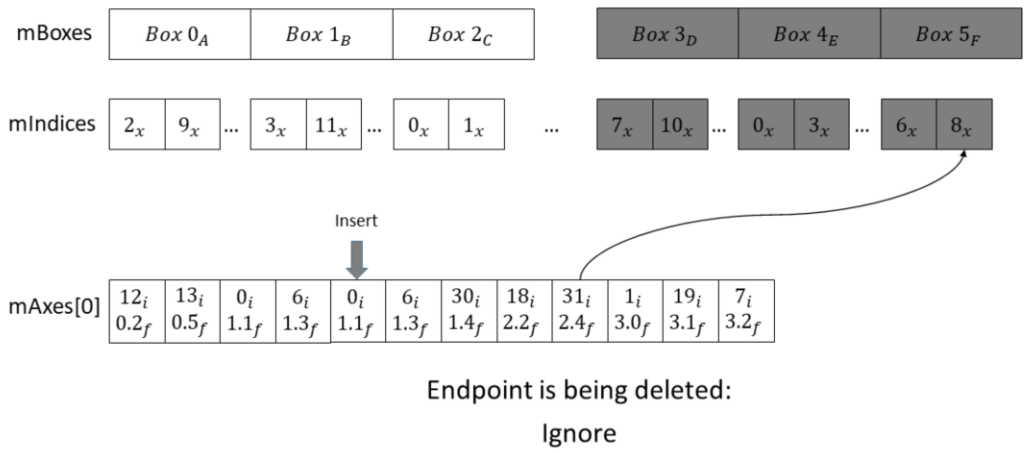
Batch Deletion – EndPoint Removal



Endpoint is being deleted:

Ignore

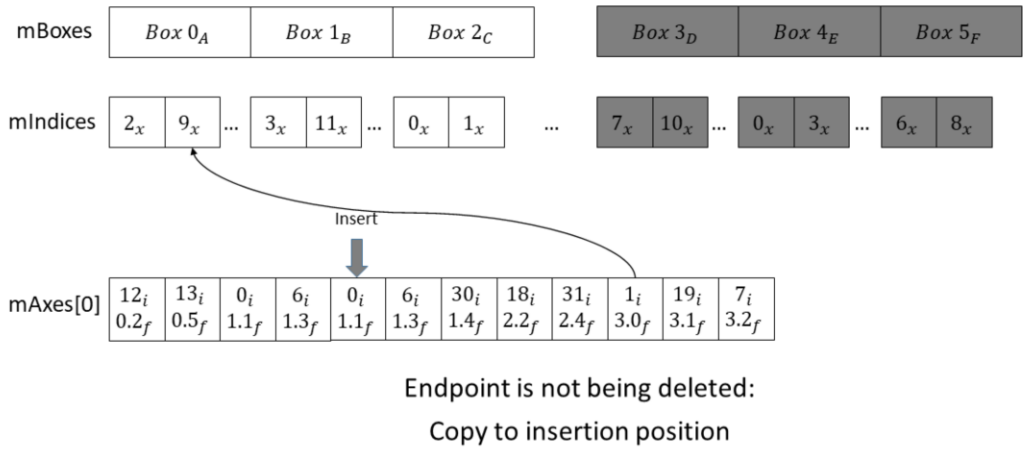
Batch Deletion – EndPoint Removal



Endpoint is being deleted:

Ignore

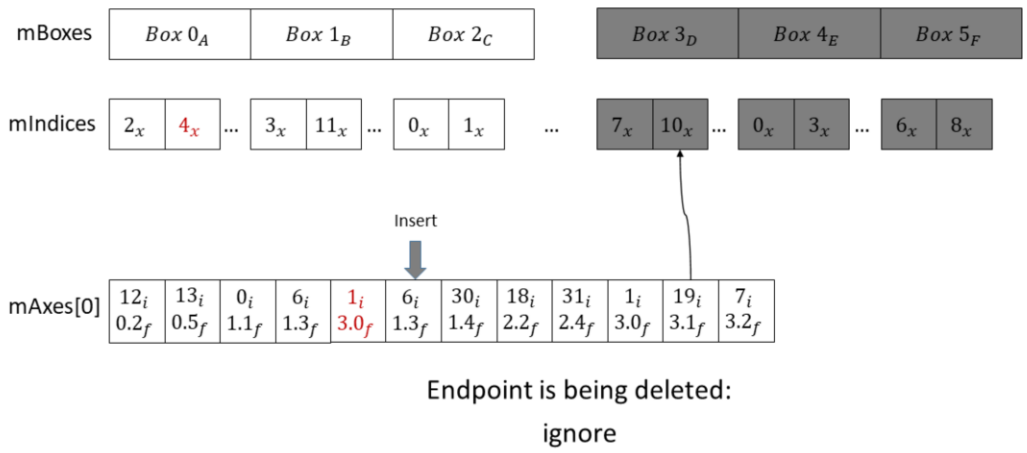
Batch Deletion – EndPoint Removal



Endpoint is not being deleted:

Copy to insertion position

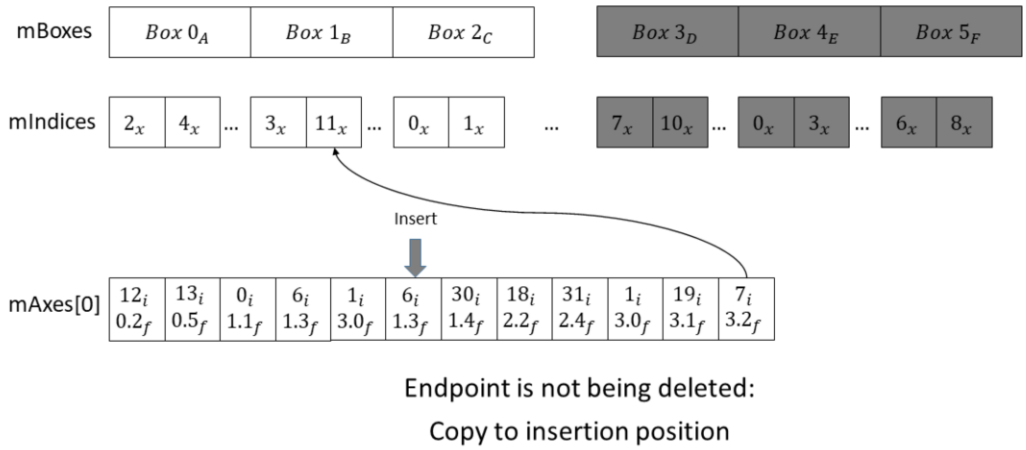
Batch Deletion – EndPoint Removal



Endpoint is being deleted:

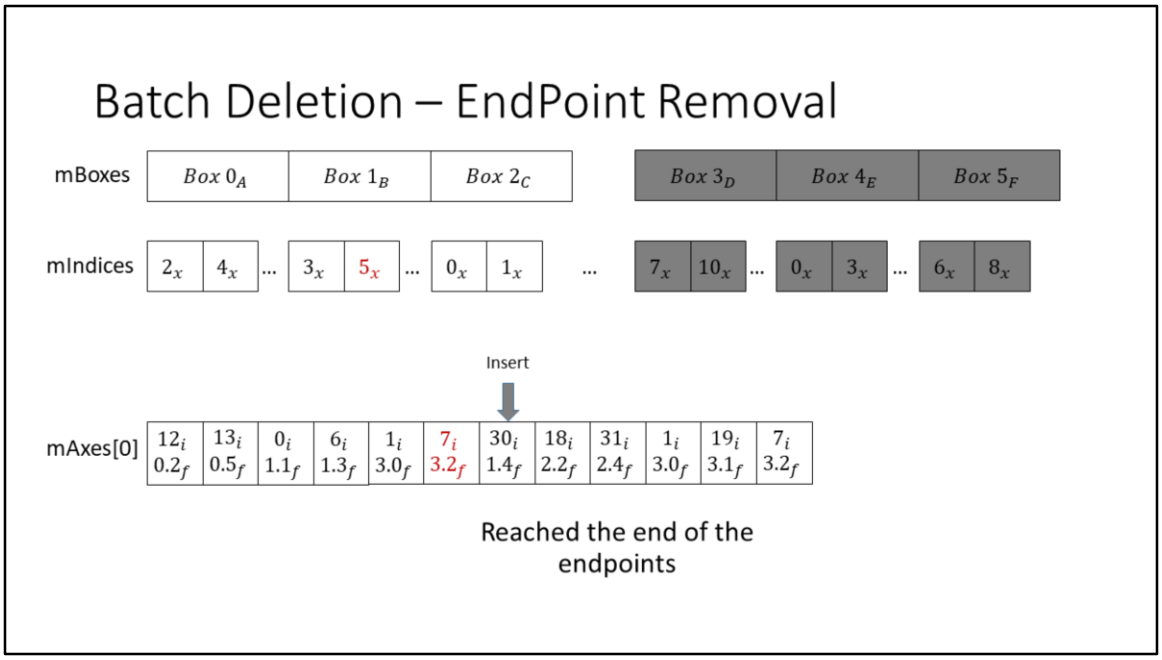
ignore

Batch Deletion – EndPoint Removal

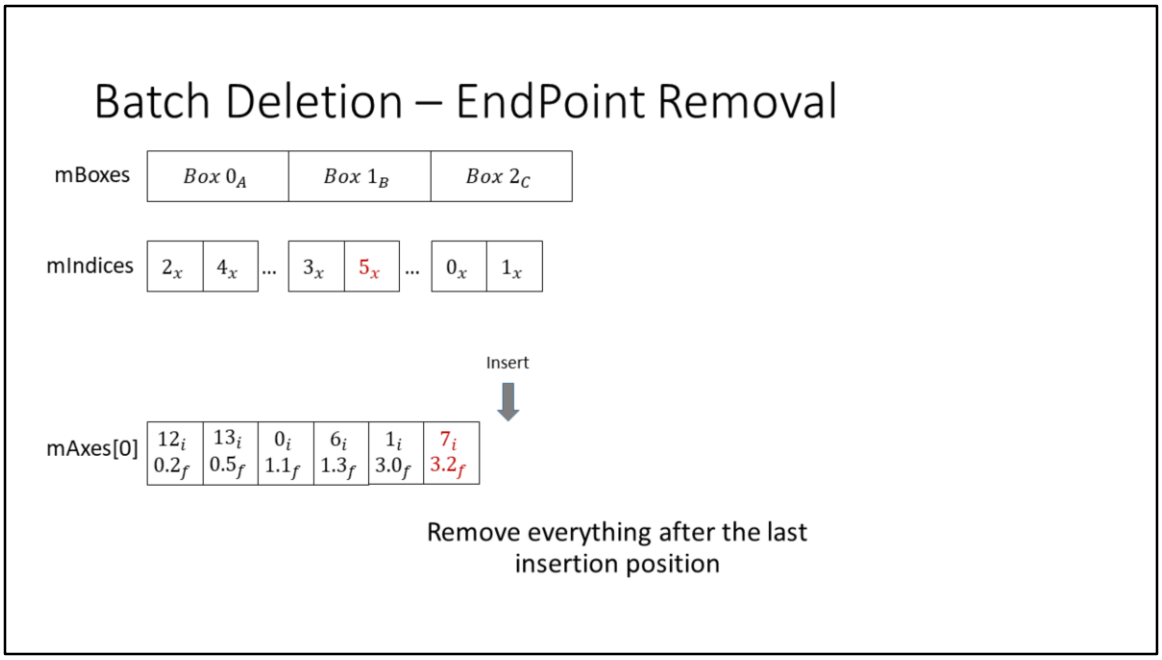


Endpoint is not being deleted:

Copy to insertion position



We terminate once the test position reaches the end of the endpoint array.



To finish off we “remove: the boxes and indices (we might just be ignoring them and marking them in a free list). Finally the endpoints at the insertion position and after are garbage we don’t care about so we simply chop off the end of the array.

Quantization

Integer comparisons are faster than floats

*Ever so slightly (5% last I checked)

SAP can work exclusively with integer comparison (via some tricks)

There's a few extra things I want to cover about SAP. One of them is what's known as quantization. Technically, integer comparison is a little faster than float comparisons (last I checked it was around 5% slower with floats). We can actually turn all float values in SAP into integers for more efficient comparisons.

Quantization

Method 1:

Choose a world scale and re-normalize to [-int max, int max]

Problems:

Accuracy
Artificial world limitations

The simplest way to quantize is to pick some world bounds, say [-m,m] and divide the every coordinate by this bounds. This gives us values in the range of [-1, 1]. Now we can scale up by the max size of an integer to span the entire integer range.

This method isn't without issues. First of all we lose some accuracy as some values that were explicitly different will snap to artificial integer boundaries. Another issue is that we've imposed a max size on our world. This can actually be a much more obnoxious issue to deal with.

Luckily, there's a much easier way to quantize!

Quantization

Method 2: Convert floats to ints

We just need integers that sort like floats

No math necessary!

Two observations of floating point numbers:

1. Negative floats are larger than positive (sign bit)
2. Negative floats sort backwards
   - Negative's floats are larger than positive (sign bit)
   - Negative floats sort backwards

The second method to quantize is to just turn a float into an int without any real math. As long as we produce an integer that sorts in the correct order we really don't care what it becomes. We technically don't even care if we can reverse engineer the value (although we'll be able to).

To do this we have to look closely at the floating point format and observe 2 key things:

1. When treating a float like an integer, negative floats are larger than positive. This is because negative floats have the sign bit (the most significant bit) set.
2. Negative floats sort backwards, that is -1000 is larger than -1. This is because a float is only negative because of its sign bit, but otherwise is the exact same as its positive counter-part.

If we can fix these two issues without losing accuracy then we have our quantization!

Quantization

Solution:

1. If positive, set the sign bit
2. If negative, binary not the value

```
const unsigned int signBit = 1 << 31;
int FloatEncode(float val)
{
  unsigned int encodedVal = *reinterpret_cast<int*>(&val);

  // If negative, binary not the entire value
  if(encodedVal & signBit)
    encodedVal = ~encodedVal;
  // If positive just set the sign bit
  else
    encodedVal |= signBit;
  return encodedVal;
}
```

Since positives need to sort greater than negatives we just need to swap their sign bits to get them to sort correctly with respect to each other. After this positive floats will sort correct with themselves and with negative values, however negative values will still sort backwards. Instead of just flipping the sign bit of negative floats, we can actually just binary not the entire value. This will make negative floats also sort correctly against each other.

Note that with this quantization method we didn't lose any precision of floating point numbers as we just converted a 32-bit float to some special and unique 32-bit integer.

Casting

One of the biggest problems with SAP

No good optimization for any casting methods

Main reason it's not used anymore

- That and clustering

Now for the unfortunate news about SAP. Although it has a pretty bad worse-case scenario, the biggest real problem with it is that it's only good at finding pairs. Doing any kind of cast or query against it is not efficient. There's something called stabbing numbers that was used in the past, but honestly I can't find any information about it anymore. The only thing I know about it is that it adds more information to each endpoint to more efficiently jump around. Adding this information increases memory overhead which causes SAP to not perform as fast and is also incredibly complicated.

Raycasting

Raycasting through non-uniform grid is hard

I'm not going to go through it here, but ray-casting through a non-uniform grid is a lot trickier than one might think. Given a starting cell and a ray, it's not that hard to traverse efficiently through the cells that the ray goes through. Unfortunately finding which cell the ray starts in is worst-case $O(n)$ as you have to search to find what "cell" the ray starts in.

Shape Casting

No good method, just do linear pass

- Or use another spatial partition

Shape casting works even worse than ray-casting. You effectively have to rasterize a shape to the non-uniform grid and traverse all cells it overlaps. Most people I know who implemented SAP (myself included) simply did a linear pass for shape-casting (or created another spatial partition just for casts).

Questions?