Subject: Re: The continuing saga of WHERE and 2D
Posted by JD Smith on Fri, 28 Feb 2003 17:35:38 GMT
View Forum Message <> Reply to Message

On Thu, 27 Feb 2003 15:47:12 -0700, David Fanning wrote:

```
> Sean Raffuse (sean@me.wustl.edu) writes:
>
>> No, that's it.  Although here is the error
>>
>>  Tau[x_mask,y_mask,*] = default_BelowDetLimit
>> % Array has too many elements.
>>
>> Is this a memory problem?
>
> Humm. A memory problem? I don't know. 1.4 million array elements seems
> like a lot, but I can easily do this:
>
> IDL> a=FltArr(3660, 1680)
> IDL> b = randomu(seed, 1407735L) * 3600L * 1680L IDL> a[b] = 5
>
> I have a problem, however, when I add a third dimension:
>
> IDL> DelVar, a
> IDL> a = FltArr(3660, 1680, 8)
> IDL> a[b,b, *] = 5
> % Array has too many elements.
> IDL> a[b,b, 0] = 5
> % Array has too many elements.
>
> I don't know what that is. I think you are going to have to ask RSI.
>
> I do notice that if I do this:
>
> IDL> c= b[0:4999]
> IDL> a[c,c,*] = 5
>
> That my computer gets very, VERY unhappy. :-( I had to reboot to get
> some response back.
>
> I think this means there is something going on here that I don't
> understand. :-)
```

I think it's pretty simple, if subtle. When it encounters a
multi-dimensional subscript, IDL look to see if all subscript vectors
have the same dimensions. If so, it "threads the list" and constructs
indices from them on the fly as:

```
 [vec1,vec2,vec3,...] ==> vec1+vec2*n1+vec3*n1*n2+...
```

where n1,n2, etc. are the sizes of the 1st,2nd, etc. dimension of the
array being indexed. I.e., you've essentially specified a short list
of index pairs, triples, etc., of length n_elements(vec1).

If, however, any of the subscripts are unspecified or
zero-dimensional, by virtue of using a single index or one of the
higher-order range operations (e.g. `*' or '0:5'), a temporary large
array of indices has to be pre-created. Why? Because you can no
longer "thread the list". Example: if I say

```
  a[ [1,2,3] , [4,5,6], 0]
```

you might think I mean for IDL to generate a list like:

```
  a[1, 4, 0]
  a[2, 5, 0]
  a[3, 6, 0]
```

but it actually expands this to:

```
  a[1, 4, 0] a[1, 5, 0] a[1, 6, 0]
  a[2, 4, 0] a[2, 5, 0] a[2, 6, 0]
  a[3, 4, 0] a[3, 5, 0] a[3, 6, 0]
```

This can be a very important distinction when subscripting with large
index vectors.

Here's an example demonstrating this:

```
IDL> a=randomu(sd,100,100,100)
IDL> help,/memory
heap memory used:    4392171, max:    4392190, gets:     1895, frees:     1475
IDL> a[*,*,*]=1
IDL> help,/memory
heap memory used:    4392203, max:    8392260, gets:     1899, frees:     1477
IDL> print,(8392260-4392190)/4
     1000017
```

Ahah, it seems a temporary index array of 100*100*100 indices was
made. Make sense. What if we use three index vectors of the same
size?

```
IDL> a=randomu(sd,100,100,100)
IDL> r1=randomu(sd,100) & r2=randomu(sd,100) & r3=randomu(sd,100)
IDL> help,/memory
heap memory used:    4385413, max:    4385432, gets:     1212, frees:      807
```

```
IDL> a[r1,r2,r3]=1
IDL> help,/memory
heap memory used:    4385448, max:    4386374, gets:     1218, frees:      811
```

In this case, a temporary index array was not needed; the three r
vectors were used together directly as a threaded list, and no extra
memory was used. How about:

```
IDL> a=randomu(sd,100,100,100)
IDL> help,/memory
heap memory used:    4383915, max:    4383934, gets:     1206, frees:      806
IDL> r1=randomu(sd,100) & r2=randomu(sd,100)
IDL> help,/memory
heap memory used:    4384920, max:    4384939, gets:     1211, frees:      807
IDL> a[r1,r2,*]=1
IDL> help,/memory
heap memory used:    4384954, max:    8385481, gets:     1217, frees:      811
```

It seems 100*100*100 indices where created here too. Looks right.
Now, let's stress things a bit:

```
IDL> a=randomu(sd,100,100,100)
IDL> r1=randomu(sd,1000) & r2=randomu(sd,1000) & r3=randomu(sd,1000)
IDL> help,/memory
heap memory used:    4396199, max:    4396218, gets:     1210, frees:      806
IDL> a[r1,r2,r3]=1
IDL> help,/memory
heap memory used:    4396234, max:    4404360, gets:     1216, frees:      810
```

Wait a minute, what's happening here? The subscript vectors, despite
being larger than the dimensions of the array they're accessing, are
still just being used directly, with no additional overhead required
for creating a temporary index array. The assignment to 1 occurs 1000
times.

What about:

```
IDL> a=randomu(sd,100,100,100)
IDL> r1=randomu(sd,1000) & r2=randomu(sd,1000)
IDL> help,/memory
heap memory used:    4392105, max:    4392124, gets:     1209, frees:      806
IDL> a[r1,r2,*]=1
IDL> help,/memory
heap memory used:    4400378, max:  404396266, gets:     1902, frees:     1478
IDL> print,(404396266-4392124)/4
   100001035
```

Uh oh. You can see that IDL had to pre-allocate a temporary index entry on the fly with 1000*1000*100 elements, despite the fact that it was used to index a much smaller array. The assignment to 1 occurs 100,000,000 times! Quite a difference. I can take this to the extreme:

```
IDL> a=randomu(sd,1,1,1)
IDL> r1=randomu(sd,10000L) & r2=randomu(sd,10000L)
IDL> help,/memory
heap memory used:     464107, max:     464126, gets:     1209, frees:      806
IDL> a[r1,r2,0]=1  ; long delay
IDL> help,/memory
heap memory used:     472380, max:  400504268, gets:     1902, frees:     1478
IDL> print,(400504268-464126)/4
   100010035
```

Oh my, nearly 1/2 Gb was allocated for the temporary index array just to assign a value to a *single* element (over and over again). What if I pre-build my index vector:

```
IDL> a=randomu(sd,1,1,1)
IDL> r1=randomu(sd,10000L) & r2=randomu(sd,10000L)
IDL> r=r1+1*r2
IDL> help,/memory
heap memory used:     504193, max:     504212, gets:     1211, frees:      806
IDL> a[r]=1   ; no delay
IDL> help,/memory
heap memory used:     504221, max:     544282, gets:     1215, frees:      808
```

What a difference this makes.

Bottom line? Keep in mind this duality in how IDL treats arrays as subscripts, and be very careful when mixing array subscripts with other types. If you mean for, e.g.

```
 [ [1,2], [3,4], 0 ] ==> [1,3,0], [2,4,0]
```

instead of

```
 [ [1,2], [3,4], 0 ] ==> [1,3,0], [1,4,0],
                 [2,3,0], [2,4,0]
```

Then you should use:

```
 [ [1,2], [3,4], [0,0] ]
```

or just pre-build your indices as a single index vector beforehand.

JD