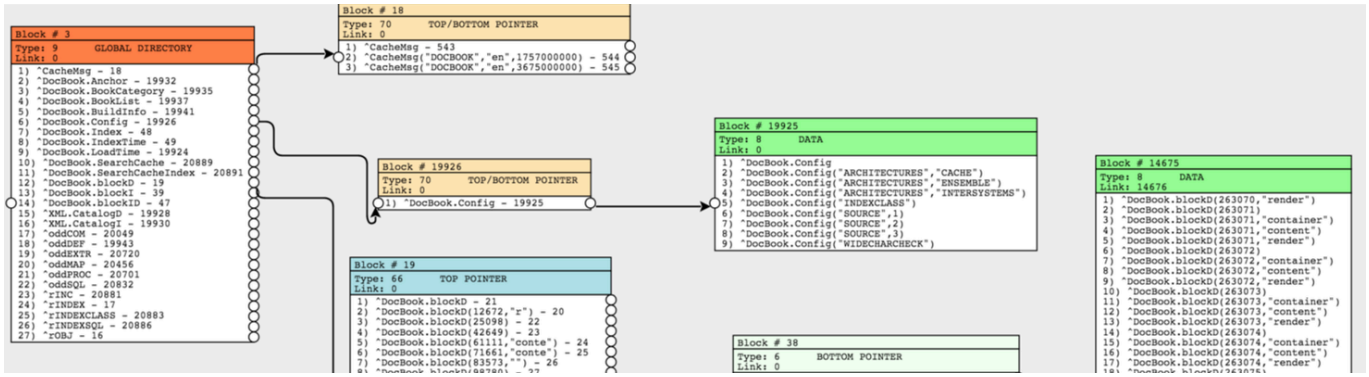
Artículo
Estevan Martinez · 30 jul, 2019 Lectura de 6 min

Estructura Interna de los Bloques de Bases de Datos en Caché (Parte 2)

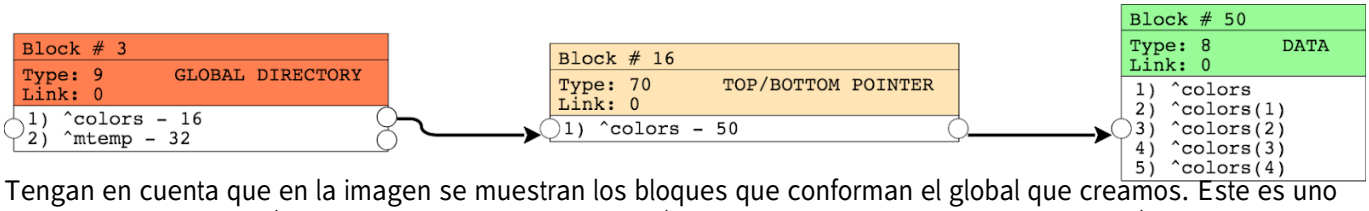
¡Hola a tod@s!

Este artículo es la continuación de mi artículo anterior, donde expliqué cómo es la estructura de una base de datos en Caché. En ese artículo describí los tipos de bloques, las conexiones que existen entre ellos y su relación con los globales. Como el artículo era completamente teórico, realicé un proyecto que ayuda a visualizar el árbol de bloques, y en este artículo explicaré su funcionamiento muy detalladamente.



Con el propósito de hacer una demostración, creé una nueva base de datos y eliminé los globales que Caché inicializa de forma predeterminada para todas las bases de datos nuevas. Crearemos un global sencillo:

```
 set ^colors(1)="red"
 set ^colors(2)="blue"
 set ^colors(3)="green"
? set ^colors(4)="yellow"
```



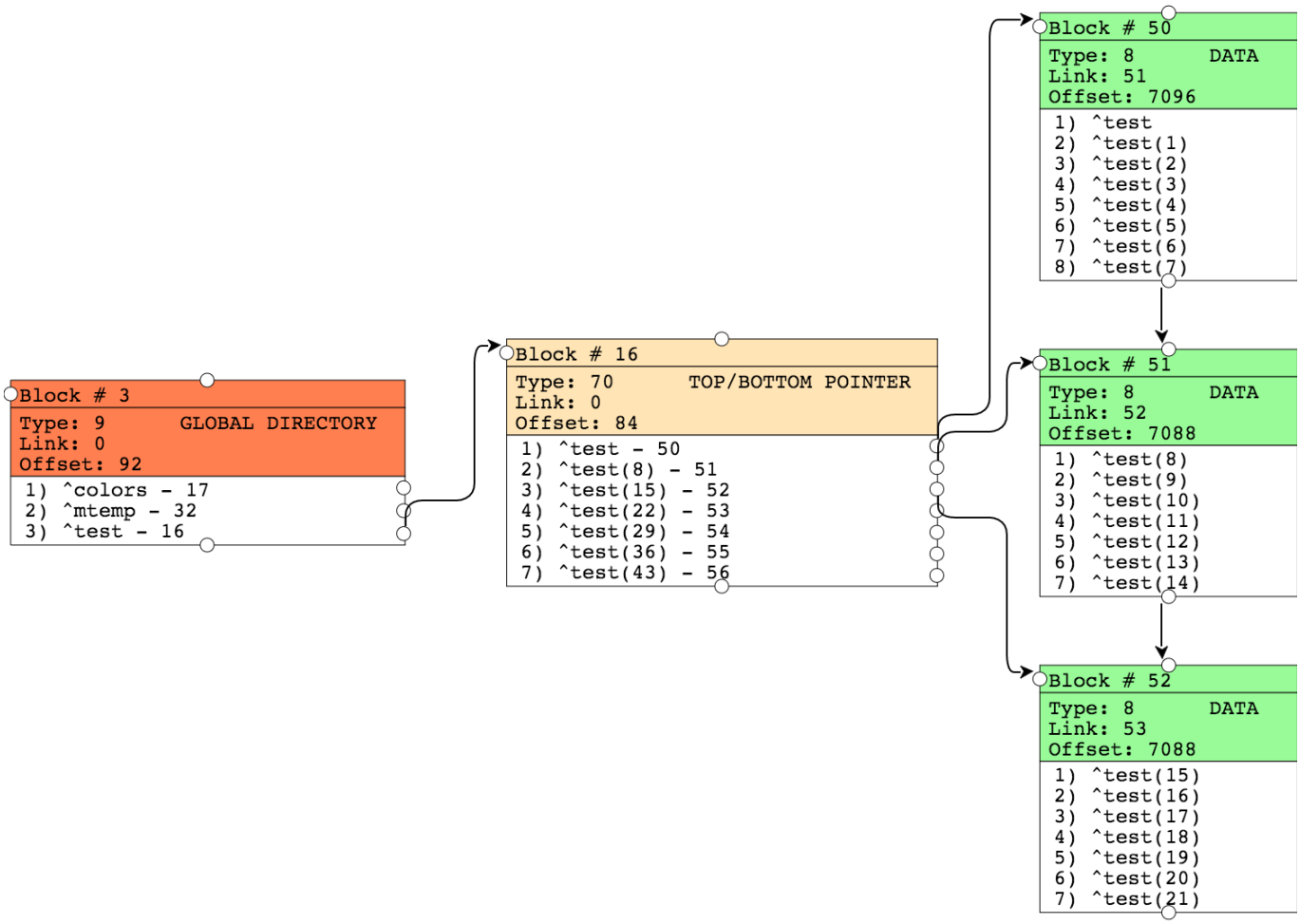
Tengan en cuenta que en la imagen se muestran los bloques que conforman el global que creamos. Este es uno sencillo, esta es la razón por la que vemos su descripción en el bloque de tipo 9 (bloque del catálogo de globales). Lo siguiente es el bloque "puntero superior e inferior" (tipo 70), ya que el árbol de globales aún no es lo suficientemente profundo, y puede utilizar un puntero para un bloque de datos que todavía cabe en un solo bloque de 8 KB.

Ahora, escribiremos tantos valores en otro global que no cabrán en un solo bloque y veremos los nuevos nodos en el bloque puntero señalando hacia los nuevos bloques de datos, que no caben en el primero.

Escribiremos 50 valores, cada valor tendrá una longitud de 1000 caracteres. Recuerde que el tamaño del bloque en nuestra base de datos es de 8192 bytes.

```
   set str=""
   for i=1:1:1000 {
       set str=str_"1"
   }
   for i=1:1:50 {
       set ^test(i)=str
   }
?   quit
```

Observen la siguiente imagen:



Tenemos bastantes nodos en el nivel del bloque puntero que señala los bloques de datos. Cada uno de los bloques de datos contiene punteros para el siguiente bloque (es el "enlace correcto"). Offset señala el número de bytes que están ocupados en este bloque de datos.

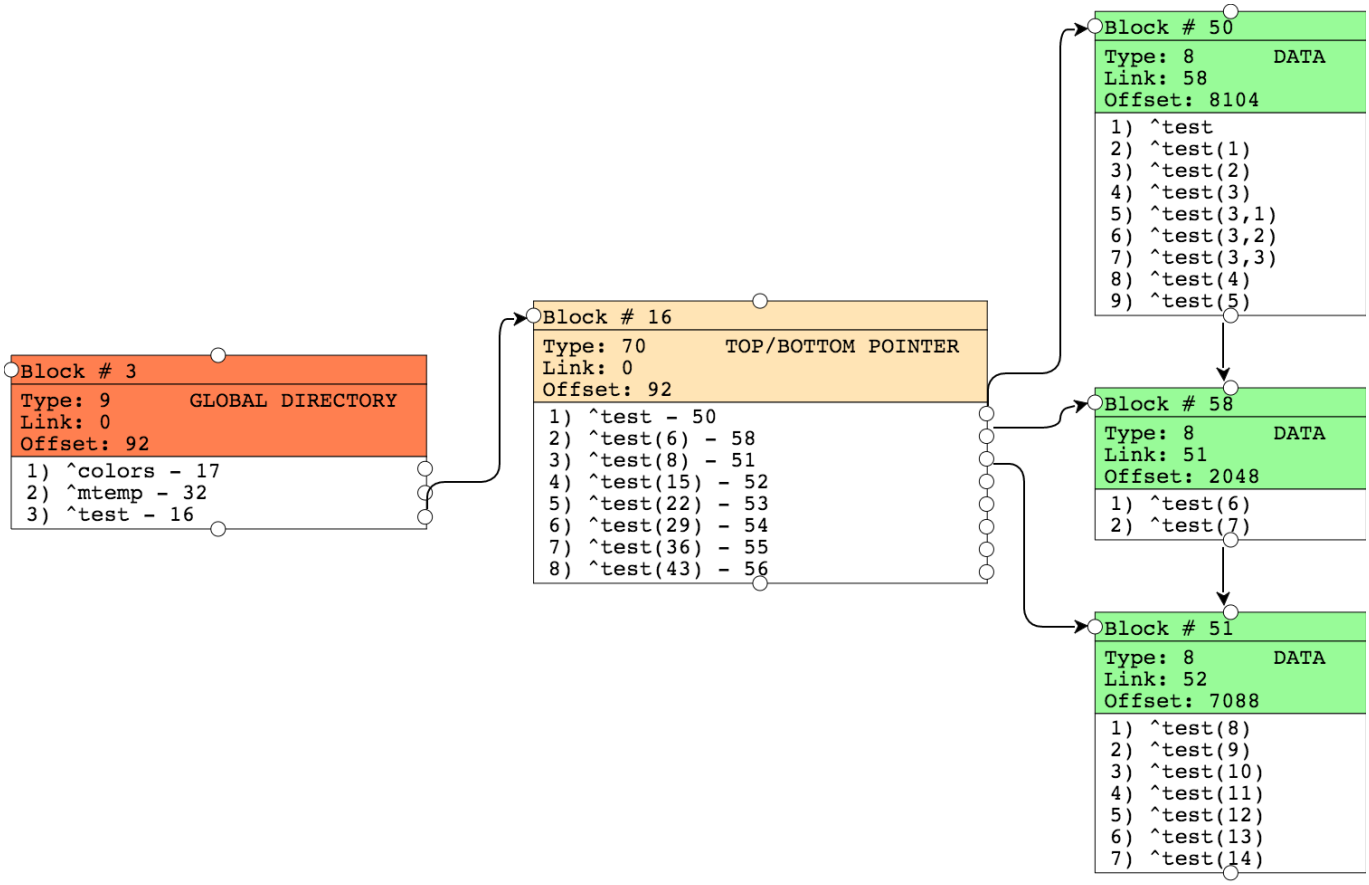
Tratemos de simular una separación de los bloques. Añadiremos tantos valores al bloque que su tamaño final superará los 8KB, lo cual provocará que el bloque se divida en dos.

Ejemplo del código:

```
   set str=""
   for i=1:1:1000 {
       set str=str_"1"
   }
   set ^test(3,1)=str
```

```
   set ^test(3,2)=str
?   set ^test(3,3)=str
```

El resultado puede verse a continuación:



El bloque 50 se dividió y ahora está lleno de datos nuevos. Los valores que se reemplazaron ahora se encuentran en el bloque 58, y en el bloque puntero ahora aparece un puntero para este bloque. Los otros bloques permanecen sin ningún cambio.

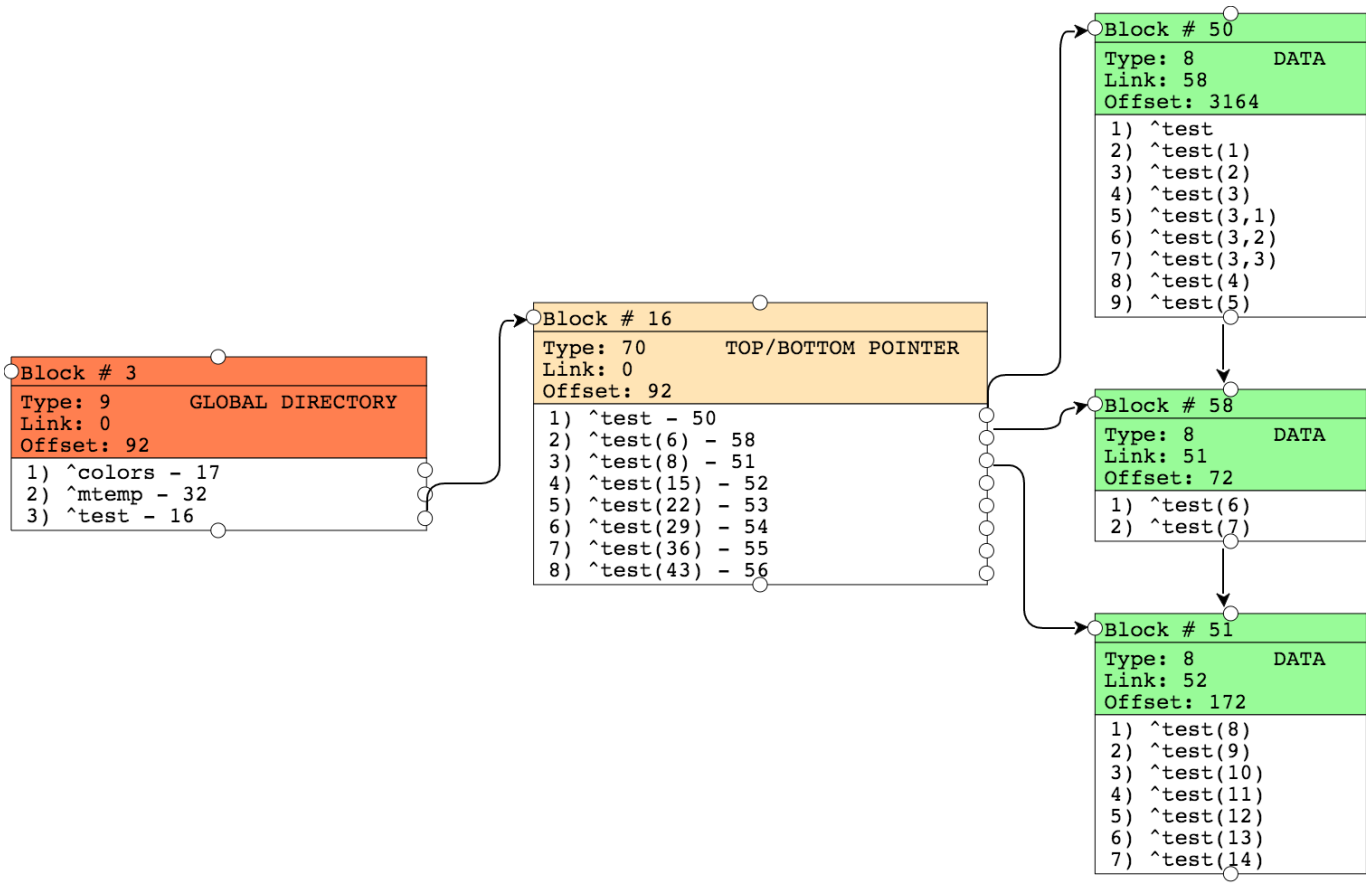
Un ejemplo con cadenas largas

Si utilizamos cadenas con más de 8KB (el tamaño del bloque de datos), obtendremos bloques de "datos grandes". Por ejemplo, podemos simular una situación de este tipo escribiendo cadenas con 10,000 bytes.

Ejemplo del código:

```
   set str=""
   for i=1:1:10000 {
       set str=str_"1"
   }
   for i=1:1:50 {
       set ^test(i)=str
?   }
```

Echemos un vistazo al resultado:



Como resultado, la estructura de los bloques en la imagen se mantuvo igual, ya que no agregamos nuevos nodos globales, sino que solo modificamos los valores. Sin embargo, el valor de Offset (el número de bytes ocupados) cambió para todos los bloques. Por ejemplo, el valor de Offset para el bloque #51 ahora es 172 en lugar de 7088. Ahora está claro que el nuevo valor no cabe en el bloque ya que, desde el puntero hasta el último byte de datos debería ser diferente pero, ¿dónde están nuestros datos? Por el momento, mi proyecto no es compatible con la posibilidad de mostrar información sobre los "bloques grandes". Utilicemos la herramienta ^REPAIR para obtener información sobre el contenido nuevo en el bloque #51.

```
Block # 51                Type: 8 DATA
Link Block: 52            Offset: 172
Count of Nodes: 7         Collate: 5              Big String Nodes: 7
Pointer Length:7          Next Pointer Length:7   Diff Byte:Hex 2B
Pointer Reference:       ^test(8)
Next Pointer Reference: ^test(15)
Next pointer stored? Yes


--more--

#     Node                    Data
1     ^test(8)                BIG: LEN 10000 BLKS: 73,74
2     ^test(9)                BIG: LEN 10000 BLKS: 75,76
3     ^test(10)               BIG: LEN 10000 BLKS: 77,78
4     ^test(11)               BIG: LEN 10000 BLKS: 79,80
5     ^test(12)               BIG: LEN 10000 BLKS: 81,82
6     ^test(13)               BIG: LEN 10000 BLKS: 83,84
7     ^test(14)               BIG: LEN 10000 BLKS: 85,86
```

Permítanme explicar con más detalle el funcionamiento de esta herramienta. Vemos un puntero que señala como el bloque correcto al #52, y el mismo número se especifica en el bloque puntero principal del siguiente nodo. La compilación de los globales se establece con el tipo 5. El número de nodos que tienen cadenas largas es 7. En

algunos casos, el bloque puede contener tanto valores de datos para algunos nodos como cadenas largas para otros, todo ello dentro de un mismo bloque. También vemos que la siguiente referencia del puntero debe esperarse al inicio del siguiente bloque.

Respecto a los bloques de cadenas largas: vemos que la palabra clave "BIG" se especifica como el valor del global. Lo que esta palabra nos indica es que los datos se almacenan en "bloques grandes". La misma línea incluye la longitud total que tiene la cadena y la lista de los bloques que almacenan este valor. Echemos un vistazo al "bloque de cadenas largas", el bloque #73.

```
Block #: 73Big string block

0000: E4 1F 00 00 18 05 00 00 4A 00 00 00 00 00 00 00        ä.......J.......
0010: 0C 7E 7F D8 00 00 00 00 00 00 00 00 31 31 31 31        .~.Ø........1111
0020: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0030: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0040: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0050: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0060: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0070: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0080: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
0090: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00A0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00B0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00C0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00D0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00E0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
00F0: 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31        1111111111111111
--more--
```

Desafortunadamente, este bloque aparece codificado. Sin embargo, podemos observar que nuestros datos se encuentran después de la información de servicio, a partir del bloque del encabezado (que siempre tiene una longitud de 28 bytes). Conocer el tipo de datos hace que la decodificación del contenido del encabezado sea bastante sencilla:

Position	Value	Description	Comment	
0-3	E4 1F 00 00	Offset pointing at the end of data	We get 8164 bytes, plus 28 bytes of the header for a total of 8192 bytes, the block is full.	
4	18	Block type	As we remember, 24 is the type identifier for long strings.	
5	05	Collate	Collate 5 stands for “ standard Caché ”	
8-11	4A 00 00 00	Right link	We get 74 here, as we remember that our value is stored in blocks 73 and 74	

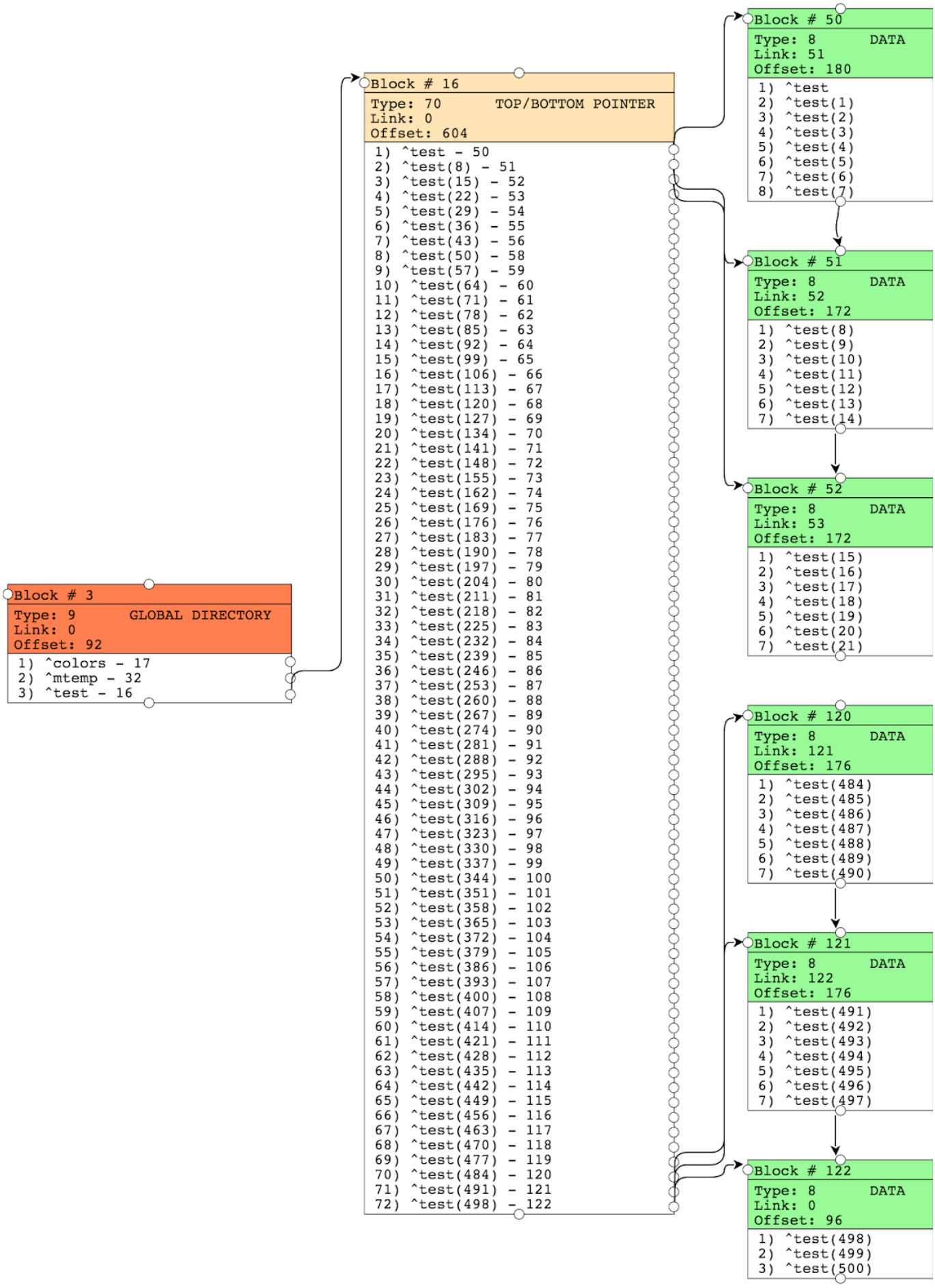
Permítanme recordarles que los datos del bloque 51 solo ocupan 172 bytes. Esto sucedió cuando guardamos valores grandes. Por ello, parece como si el bloque estuviera casi vacío con solo 172 bytes de datos útiles y sin embargo, ¡ocupa 8KB! Está claro que en una situación como esta el espacio libre se llenará de valores nuevos, pero Caché también nos permite comprimirlos como un global. Por esta razón, la clase %Library.GlobalEdit tiene

el método CompactGlobal. Para comprobar la eficacia de este método utilizaremos nuestro ejemplo con un gran volumen de datos, por ejemplo, al crear 500 nodos.

Esto es lo que obtuvimos:

```
kill ^test
  for l=1000,10000 {
      set str=""
      for i=1:1:l {
          set str=str_"1"
      }
      for i=1:1:500 {
          set ^test(i)=str
      }
  }
  quit
```

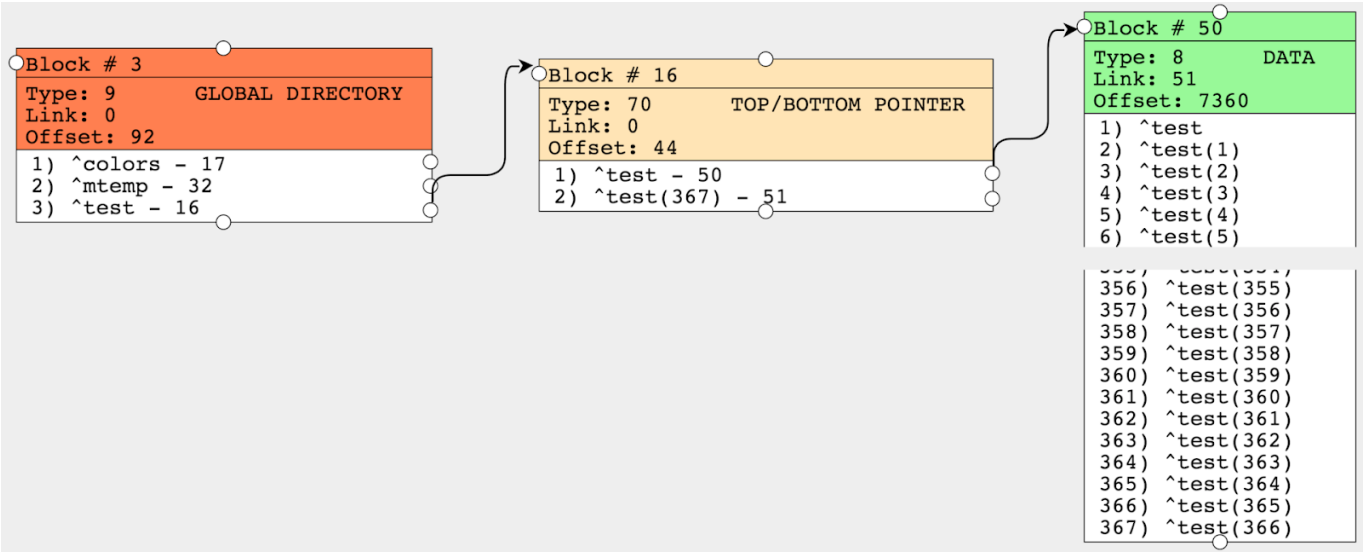
En la siguiente imagen no se muestran todos los bloques, pero este punto debe quedar claro. Tenemos muchos bloques de datos, pero con un menor número de nodos.



Si ejecutamos el método CompactGlobal:

```
write ##class(%GlobalEdit).CompactGlobal("test","c:\intersystems\ensemble\mgr\test")
```

Echemos un vistazo al resultado. El bloque puntero ahora solamente tiene 2 nodos, lo cual significa que todos nuestros valores se encuentran en esos dos nodos, mientras que inicialmente teníamos 72 nodos en el bloque puntero. Por lo tanto, nos deshicimos de 70 nodos y así redujimos el tiempo de acceso hacia los datos cuando pasan por el global, ya que se necesitan menos operaciones para leer los bloques.



El método CompactGlobal acepta varios parámetros, como el nombre del global, la base de datos y el valor que indica la capacidad de llenado, el 90% lo hace de forma predeterminada. Y ahora vemos que Offset (el número de bytes ocupados) es igual a 7360, lo cual se encuentra alrededor de ese 90%. Algunos parámetros de salida de la función: el número de megabytes procesados y el número de megabytes después de la compresión. Anteriormente, los globales se comprimían con ayuda de la herramienta ^GCOMPACT, que ahora se considera obsoleta.

Cabe destacar que una situación donde los bloques solo se llenan parcialmente es bastante normal. Por otra parte, en ocasiones es posible que no deseemos comprimir los globales. Por ejemplo, si su global se leyó casi completamente y tuvo pocas modificaciones, la compresión puede ser útil. Pero si el global se modifica todo el tiempo, cierta escasez en los bloques de datos evita el problema de tener que dividir los bloques con mucha frecuencia, y el almacenamiento de los nuevos datos será más rápido.

En la siguiente parte de este artículo revisaré otra característica de mi proyecto, la cual se implementó durante el primer hackatón que se realizó en la escuela de InterSystems de 2015, un diagrama de distribución para los bloques en las bases de datos y su aplicación práctica.

¡Espero que les haya resultado útil!

#Bases de datos #Administración del sistema #Caché

URL de fuente:https://es.community.intersystems.com/post/estructura-interna-de-los-bloques-de-bases-de-datos-en-cach%C3%A9-parte-2