Understanding NTFS Compression

ntdebug 20 May 2008 12:04 PM | 5

As our group's file systems expert, I see most of the common problems associated with the use/misuse of NTFS compression. Before you check the "Compress contents to save disk space" checkbox, it might be good to understand how this affects the happy little bits, bytes, & nibbles running around on the disk.

Often backup applications receive ERROR_DISK_FULL errors attempting to back up compressed files and this causes quite a bit of confusion when there are still several gigabytes of free space on the drive. Other issues may also occur when copying compressed files. The goal of this blog is to give the reader a more thorough understanding of what really happens when you compress NTFS files.

Compression Units…

NTFS uses a parameter called the "compression unit" to define the granularity and alignment of compressed byte ranges within a data stream. The size of the compression unit is based entirely on NTFS **cluster size** (refer to the table below for details). In the descriptions below, the abbreviation "CU" is used to describe a Compression Unit and/or its size.

The default size of the CU is 16 clusters, although the actual size of the CU really depends on the cluster size of the disk. Below is a chart showing the CU sizes that correspond to each of the valid NTFS cluster sizes.

Cluster Size	Compression Unit	Compression Unit (hex bytes)
512 Bytes	8 KB	0x2000
1 KB	16 KB	0x4000
2 KB	32 KB	0x8000
4 KB	64 KB	0x10000
8 KB	64 KB	0x10000
16 KB	64 KB	0x10000
32 KB	64 KB	0x10000
64 KB	64 KB	0x10000

Native NTFS compression does not function on volumes where the cluster size is greater than 4KB, but sparse file compression can still be used.

NTFS Sparse Files…

The **Sparse Files** features of NTFS allow applications to create very large files consisting mostly of zeroed ranges without actually allocating **LCN**s (logical clusters) for the zeroed ranges.

For the code-heads in the audience, this can be done by calling DeviceIoControl with the FSCTL_SET_SPARSE IO control code as shown below.

```
BOOL SetSparse(HANDLE hFile)
     {
           DWORD Bytes;
           return DeviceIoControl(hFile, FSCTL_SET_SPARSE, NULL, 0, NULL, 0, &Bytes, NULL);
     }
```

To specify a zeroed range, your application must then call the DeviceIoControl with the FSCTL_SET_ZERO_DATA IO control code.

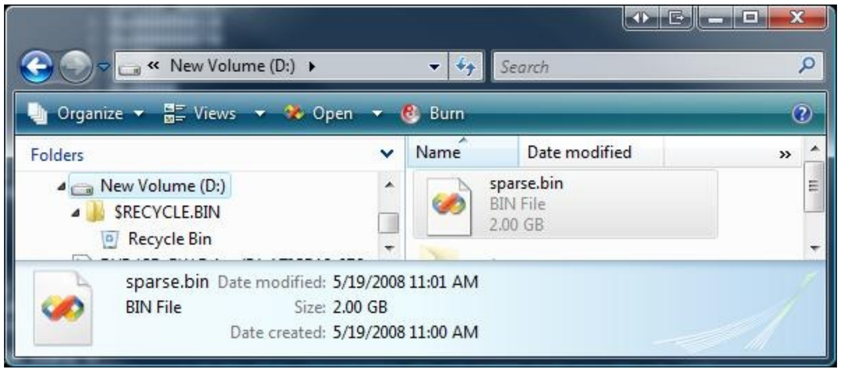
```
BOOL ZeroRange(HANDLE hFile, LARGE_INTEGER RangeStart, LONGLONG RangeLength)
     {
            FILE_ZERO_DATA_INFORMATION  FileZeroData;
            DWORD Bytes;
            FileZeroData.FileOffset.QuadPart = RangeStart.QuadPart;
            FileZeroData.BeyondFinalZero.QuadPart = RangeStart.QuadPart + RangeLength + 1;
            return DeviceIoControl(  hFile,
                                     FSCTL_SET_ZERO_DATA,
                                     &FileZeroData,
                                     sizeof(FILE_ZERO_DATA_INFORMATION),
                                     NULL,
                                     0,
                                     &Bytes,
                                     NULL);
     }
```

Because sparse files don't actually allocate space for zeroed ranges, a sparse file can be larger than the parent volume. To do this, NTFS creates a placeholder VCN (virtual cluster number) range with no logical clusters mapped to them.

Any attempt to access a "sparsed" range would result in NTFS returning a buffer full of zeroes. Accessing an allocated range would result in a normal read of the allocated range. When data is written to a sparse file, an allocated range is created which is exactly aligned with the compression unit boundaries containing the byte(s) written. Refer to the example below. If a single byte write occurs for virtual cluster number VCN 0x3a, then all of Compression Unit 3 (VCN 0x30 - 0x3f) would become an allocated LCN (logical cluster number) range. The allocated LCN range would be filled with zeroes and the single byte would be written to the appropriate byte offset within the target LCN.

```
[...] - ALLOCATED
(,,,) - Compressed
{   } - Sparse (or free) range
   / 00000000000000000000000000000000000000000000000000000000000000000000000000000000
VCN  00000000000000001111111111111111222222222222222233333333333333334444444444444444
   \ 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
          CU0             CU1             CU2             CU3             CU4
     |++++++++++++++||++++++++++++++||++++++++++++++||++++++++++++++||++++++++++++++|
     {                                              }[..............]{              }
Extents  {
    VCN = 0x0 LEN = 0x30                        CU0 - CU2
    VCN = 0x30 LEN = 0x10: LCN = 0x2a21f        CU3
    VCN = 0x10 LEN = 0x10                       CU4
}
```

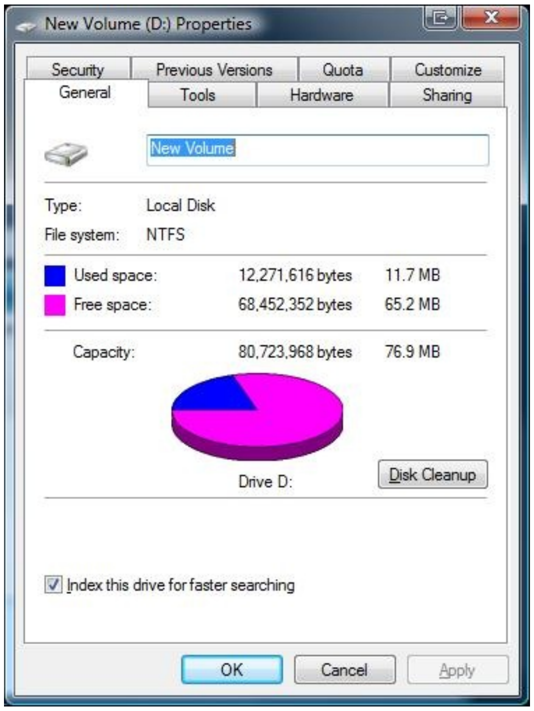
Below is a screen shot of a 2GB file that was created using the sparse file API's.



sparse.bin
BIN File
2.00 GB

sparse.bin Date modified: 5/19/2008 11:01 AM
BIN File Size: 2.00 GB
Date created: 5/19/2008 11:00 AM

Notice that this volume is only 76.9 MB, yet it has a 2 GB file in the root folder.



New Volume

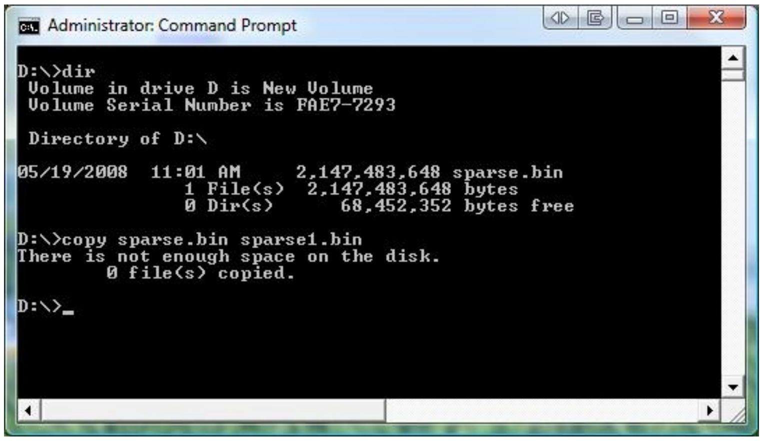
Type: Local Disk
File system: NTFS

🟦 Used space: 12,271,616 bytes 11.7 MB
🟪 Free space: 68,452,352 bytes 65.2 MB

Capacity: 80,723,968 bytes 76.9 MB

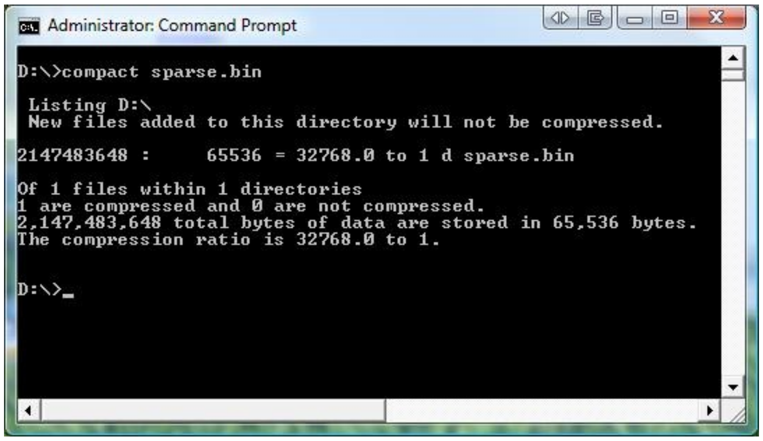
[✓] Index this drive for faster searching

If we attempt to duplicate this sparse file with the COPY command, it fails. This is because COPY doesn't know how to duplicate the sparse attributes on the file, so it attempts to create a real 2GB file in the root of D:. This can occur in production when attempting to move a large database file from one volume to another. If you have a database application that uses sparse attributes, then it is a good practice to use the database software's backup / restore features when moving the database to a different volume.



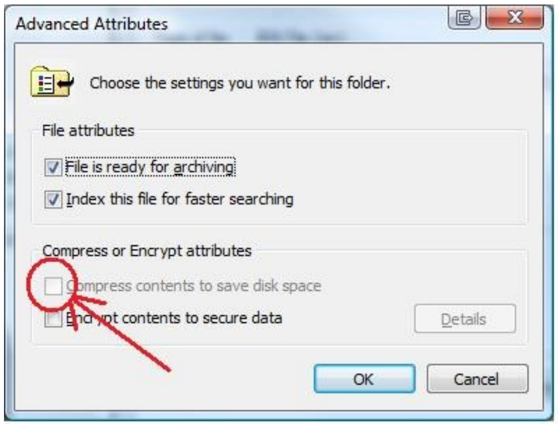
```
D:\>dir
 Volume in drive D is New Volume
 Volume Serial Number is FAE7-7293

 Directory of D:\

05/19/2008  11:01 AM     2,147,483,648 sparse.bin
               1 File(s)  2,147,483,648 bytes
               0 Dir(s)      68,452,352 bytes free

D:\>copy sparse.bin sparse1.bin
There is not enough space on the disk.
        0 file(s) copied.

D:\>_
```

Now let's look at the file's properties with the COMPACT utility. Notice that the file shows up as compressed, and it has a huge compression ratio.



```
D:\>compact sparse.bin

 Listing D:\
 New files added to this directory will not be compressed.

2147483648 :      65536 = 32768.0 to 1 d sparse.bin

Of 1 files within 1 directories
1 are compressed and 0 are not compressed.
2,147,483,648 total bytes of data are stored in 65,536 bytes.
The compression ratio is 32768.0 to 1.

D:\>_
```

If you go back and look at the file properties from EXPLORER, you will notice that there is no compression checkbox (or any other indication that the file is compressed). This is because the shell does not check for the sparse bit on the file.


Advanced Attributes
Choose the settings you want for this folder.
File attributes
Compress or Encrypt attributes
Details
OK
Cancel

[✓] File is ready for archiving
[✓] Index this file for faster searching
[ ] Compress contents to save disk space
[ ] Encrypt contents to secure data

In short, use caution when moving sparse files from one location to another. Applications tell the file system the offsets of zeroed ranges, so you should always leave the management of sparse files to the application that created them. Moving or copying the sparse files manually may cause unexpected results.

NTFS Compression…

Now that we have discussed sparse files, we will move on to conventional NTFS compression.

NTFS compresses files by dividing the data stream into CU's (this is similar to how sparse files work). When the stream contents are created or changed, each CU in the data stream is compressed individually. If the compression results in a reduction by one or more clusters, the compressed unit will be written to disk in its compressed format. Then a sparse VCN range is tacked to the end of the compressed VCN range for alignment purposes (as shown in the example below). If the data does not compress enough to reduce the size by one cluster, then the entire CU is written to disk in its uncompressed form.

This design makes random access very fast since only one CU needs to be decompressed in order to access any single VCN in the file. Unfortunately, large sequential access will be relatively slower since decompression of many CU's is required to do sequential operations (such as backups).

In the example below, the compressed file consists of six sets of mapping pairs (encoded file extents). Three allocated ranges co-exist with three sparse ranges. The purpose of the sparse ranges is to maintain VCN alignment on compression unit boundaries. This prevents NTFS from having to decompress the entire file if a user wants to read a small byte range within the file. The first compression unit (CU0) is compressed by 12.5% (which makes the allocated range smaller by 2 VCNs). An additional free VCN range is added to the file extents to act as a placeholder for the freed LCNs at the tail of the CU. The second allocated compression unit (CU1) is similar to the first except that the CU compressed by roughly 50%.

NTFS was unable to compress CU2 and CU3, but part of CU4 was compressible by 69%. For this reason, CU2 & CU3 are left uncompressed while CU4 is compressed from VCNs 0x40 to 0x44. Thus, CU2, CU3, and CU4 are a single run, but the run contains a mixture of compressed & uncompressed VCNs.

NOTE: Each set of brackets represents a contiguous run of allocated or free space. One set of NTFS mapping pairs describes each set of brackets.

```
[...] - ALLOCATED
(,,,) - Compressed
{   } - Sparse (or free) range
   / 00000000000000000000000000000000000000000000000000000000000000000000000000000000
VCN  00000000000000001111111111111111222222222222222233333333333333334444444444444444
   \ 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
           CU0             CU1             CU2             CU3             CU4
     |++++++++++++++||++++++++++++++||++++++++++++++||++++++++++++++||++++++++++++++|
     (,,,,,,,,,,,,){}(,,,,,,){      }[...............................,,,,){         }
Extents  {
    VCN = 0x0 LEN = 0xe : LCN = 0x29e32d        CU0
    VCN = 0xe LEN = 0x2                         CU0
    VCN = 0x10 LEN = 0x8 : LCN = 0x2a291f       CU1
    VCN = 0x18 LEN = 0x8                        CU1
    VCN = 0x20 LEN = 0x25 : LCN = 0x39dd49      CU2 - CU4
    VCN = 0x45 LEN = 0xb                        CU4
}
```

Now we come to the part where we describe limitations of this design. Below are some examples of what happens when things go wrong while reading / writing compressed files.

Disk full error during a backup read operation or file copy…

1. NTFS determines which compression unit is being accessed.
2. The compression unit's entire allocated range is read.
3. If the unit is not compressed, then we skip to step 5. Otherwise, NTFS would attempt to reserve (but not allocate) the space required to write the decompressed CU back to disk. If insufficient free space exists on the disk, then the application might get an ERROR_DISK_FULL during the read.
4. The CU would be decompressed in memory.
5. The decompressed byte range would be mapped into cache and returned to the requesting application.
6. If part of the CU is altered in cache…
    a. The reserved disk space from step 3 would become allocated space.

b. The CU contents would be compressed and flushed back to the newly allocated region (the LCN location will usually not change).
c. Any recoverable disk space within the CU would be freed.

Failure to copy a large file to a compressed folder…

This is the most common problem seen with compression, and currently the solution is to educate users about limitations. NTFS compression creates approximately one file fragment for every 16 clusters of data. Because the max cluster size allowed for standard compression is 4K, the largest compression unit allowed is 64KB. In order to compress a 100 GB file into 64KB sections, you could potentially end up with 1,638,400 fragments. Encoding 1,638,400 fragments becomes problematic for the file system and can cause a failure to create the compressed file. On Vista and later, the file copy will fail with STATUS_FILE_SYSTEM_LIMITATION. On earlier releases, the status code would be STATUS_INSUFFICIENT_RESOURCES. If possible, avoid using compression on files that are large, or critical to system performance.

I received feedback from the NTFS Principal Development Lead about this blog. Fortunately, most of the feedback was good, but he asked that I add a maximum size recommendation. According to our development team's research, 50-60 GB is a "reasonable size" for a compressed file on a volume with a 4KB cluster size. This "reasonable size" goes down sharply for volumes with smaller cluster sizes.

Compressing .VHD (Virtual Hard Disk) files causes slow virtual machine performance…

A popular hammer company received a call from a customer complaining "It sure hurts when I hit my thumb with your hammers!" Likewise, if you compress .VHD files, it is going to be a painful experience, so please compress responsibly.

Best Regards,

Dennis Middleton *"The NTFS Doctor"*

Comments

Anon
7 Jul 2008 11:29 AM

So you say that compressed files 'hurt' VHD performance significantly. Does this also follow true for NTFS sparse files?

I could believe the implementation for sparse files is lighter (RLE) and therefore quicker to return, but I suspect there's a performance hit caused by fragmentation when you have to expand the file? In which case is the performance penalty from the decompression or mostly from the file fragmentation (which could be corrected)?

[A sparse VHD is an interesting scenario and I would like to test that when I have a few free cycles.

In the meanwhile, here's what I would expect to find. Each random access to the physical device takes between 3-10ms (disregarding hardware cache). Because of this, performance hinges on how many random access reads & writes are necessary to/from the device. For example, hunting through the volume looking for 1 thousand 4KB fragments will be much more costly than a single contiguous 4MB read. The storage stack may break up the 4MB contiguous read into max read units (depending on hardware & storage driver requirements), but it is still going to be much faster than the 1,000 individual reads.

If you step through a sparse file adding a few bytes to every other compression unit, you could quickly find yourself with a million fragments (and a very messy fragmented file). The host has very little control over how the guest OS allocates & frees disk space on the VHD, so I suspect that you would initially have good performance, but the VHD access would become increasingly slow over time.

Regards,
Dennis]

law
30 Aug 2008 2:55 PM

So suppose I am writing a file to disk (or modify data from a "CU") -- at what point does NT try to recompress the the data to try to save space? Each time the data would be written from a "CU"-size multiple from memory to the physical disk?

[That is absolutely correct.]

Reiner Saddey
8 Mar 2009 5:49 AM

I just want to thank you for your in-depth explanation of how NTFS compression is bound to create fragmentation.

I'm using Ntbackup to create backup sets within NTFS compressed folders and just wondered if and how fragmentation might be avoided.

Having read your article, I now know that short of running Defrag there's no way to avoid fragments. They should thus be considered a feature resulting from the space left by shrinking CUs.

Still I suspect that it might be possible for the compression to create contiguous file allocations in the first place provided those files are written sequentially. In other words, if the creation of a sparse file would proceed completely sequentially and if the compression would produce CUs in order form sequential application writes at EOF, contiguous allocation should be feasible without running Defrag.

Again, thanks to your team and keep up your great work,

Reiner

[Hi Reiner, DNA molecules describe the type, shape, and arrangement of cells within an organism. Mapping pairs are similar, but they describe the type, shape, and arrangement of clusters within a stream. Normally when you defragment an uncompressed file, the defrag utility rearranges the many fragments into one (or more) larger contiguous regions. When the allocations become contiguous, we can calculate a less complex DNA chain for the stream. Copy your file into a compressed folder and now your file has a new type of DNA with additional requirements. It would be possible to relocate the allocated portions of the CU's into a logically contiguous space, but the number of DNA chains would remain the same. We must have one VCN->LCN mapping for each compressed allocation plus a VCN placeholder for any sparse regions. The VCN placeholders are required for alignment purposes, so they must be present regardless of how the allocations line up on disk. The end result is that the CU's would be packed closer together, but the number of mapping pairs required to describe the CU's would be the same. I hope this helps! .]

reader
2 Sep 2012 11:53 AM

Hi,

there is an article on tomshardware that states using NTFS compression will increase SSD write amplification, and therefore, speed-up flash memory wearout. However, there was no raw data to prove this. Could you comment?

-------quote--------------

With information being compressed on the fly, you're consuming more of an SSD's available write cycles than if you were writing the files uncompressed.

www.tomshardware.com/reviews/ssd-ntfs-compression,3073-11.html

----

Klimax
22 Oct 2012 2:23 AM

In theory possible. If SSD uses compression (like all Sandforce based) and if that compression is better then NTFS compression then the controller might not be able to use all tricks to reduce amount of data to be written.