vmware®

Technical Note

Virtual Disk Format 5.0

VMware ESXi and Hosted Products

The document describes the virtual machine disk (VMDK) format and contains the following sections:

- "Virtual Disks for Virtual Machines" on page 1
- "Layout Basics" on page 2
- "The Descriptor File" on page 3
- "Simple Extents" on page 5
- "ESXi Host Sparse Extents" on page 9
- "Stream-Optimized Compressed" on page 11
- "Glossary" on page 14

Virtual Disks for Virtual Machines

When a virtual machine's operating system reads and writes to virtual disk, it uses the same interfaces as for physical disk. VMware designed the VMDK (virtual machine disk) format to mimic the operation of physical disk. Virtual disks are stored as one or more VMDK files on the host computer or remote storage device, and appear to the guest operating system as standard disk drives.

VMware platform products all support the VMDK format, with slight variations.

Hosted platform products such as VMware Workstation or VMware Fusion store VMDK files on a file system provided by an underlying host operating system, either Windows, Linux, or Mac OS X.

Datacenter platform products store VMDK files either on the local storage of an ESXi host, or on a network connected storage device. On ESXi hosts, VMDK files are usually stored on VMFS (virtual machine file system) partitions, optimized for large-file storage, but can also be stored on NAS partitions (NFS).

VMFS-3 was introduced in ESX 3.0 and still supported for ESX/ESXi 4.0 and 4.1. VMFS-4 was never released. VMware introduced VMFS-5 in vSphere 5, with enhancements shown in Table 1.

Table 1. Comparison of VMFS-3 and VMFS-5

VMFS-3	VMFS-5
The largest extent for a disk volume was 2TB.	Extent limit increased to ~60TB, for larger volumes including RDM.
MBR (master boot record) partition type.	GPT (GUID partition table) supports larger extents.
Block size was 1, 2, 4, or 8MB for very large files.	Unified 1MB block size supports very large files > 256GB.
Smallest sub-block was 64KB.	8KB sub-block so small files consume less space and grow easily.
Maximum file count was 30,720.	Support for > 100,000 files per volume.
Maximum VMDK file size is 2TB.	Same limit.
Maximum number of supported LUNs is 256.	Same limit.
Locking of entire LUN by SCSI reservation.	Per-sector VAAI hardware-assisted locking reduces disk contention.

Information in this technical note applies to virtual disks created on Workstation 5 or later, VMware Fusion, VMware Server, ESX 3.0 or later, and ESXi 3.5 or later. Earlier products may use formats different from the ones described here. Topics that are not discussed in this document include the following:

- Virtual disks created on ESX 2 hosts or earlier, GSX 3 or earlier, Workstation 4 or earlier, or VMware ACE. Also virtual disks created in legacy mode on Workstation 5.
- Device-backed virtual disks.
- Encryption, including encrypted extents and encrypted descriptor files.
- Defragmenting, shrinking, and consolidating of virtual disks.

This technical note proceeds with a high-level introduction to the layout of the files that make up a VMware virtual disk. It then drills down into details of the data structures inside those virtual disk files.

Layout Basics

VMware virtual disks can be described at a high level by looking at two key characteristics:

- The virtual disk may use backing storage contained in a single file, or it may use storage that consists of a collection of smaller files.
- All of the disk space needed for a virtual disk's files may be allocated at the time the virtual disk is created, or the virtual disk may start small and grow as needed to accommodate new data.

A particular virtual disk may have any combination of these two characteristics.

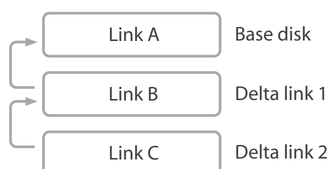
One characteristic of recent-generation VMware virtual disks is that a text descriptor describes the layout of the data in the virtual disk. This descriptor may be saved as a separate file or may be embedded in a file that is part of a virtual disk. The section titled "The Descriptor File" on page 3 explains the information contained in the descriptor.

The way a virtual disk uses storage space on a physical disk varies, depending on the type of virtual disk you select when you create the virtual machine.

Initially, for example, a virtual disk consists of only the base disk. If you take a snapshot of a virtual machine, its virtual disk includes both the base link and a delta link (referred to in some product documentation as a redo-log file). When the guest operating system writes to disk, changes since you took the snapshot are stored in the delta link. It is possible for more than one delta link to be associated with a particular base disk.

You can think of the base disk and the delta links as links in a chain. The virtual disk consists of all the links in the disk chain.

Figure 1. Links in the chain comprise the virtual disk



Each link in the chain is made up of one or more extents.

Figure 2. Extents that make up a link

An extent is a region of physical storage, often a file, that is used by the virtual disk.

In the links diagram above, links B and C are necessarily made up of extents that begin small and grow over time, referred to as sparse extents. Link A can be made up of extents of any kind – sparse, preallocated, or even backed directly by a physical device.

The Descriptor File

For a more detailed view of how these elements of a virtual disk come together in practice, look at the following example text descriptor file, called `test.vmdk`. It describes a link in a virtual disk that is split into files no larger than 2GB each and that starts small and grows as data is added.

Contents of the descriptor file are not case-sensitive. Lines beginning with # are comments and are ignored by the VMware program that opens the disk.

```
% cat test.vmdk
# Disk DescriptorFile
version=1
               CID=fffffffe
               parentCID=ffffffff
               createType="twoGbMaxExtentSparse"
# Extent description
               RW 4192256 SPARSE "test-s001.vmdk"
               RW 4192256 SPARSE "test-s002.vmdk"
               RW 2101248 SPARSE "test-s003.vmdk"
# The Disk Data Base
               #DDB
               ddb.adapterType = "ide"
               ddb.geometry.sectors = "63"
               ddb.geometry.heads = "16"
               ddb.geometry.cylinders = "10402"
```

The Header

The first section of the descriptor is the header. It provides the following information about the virtual disk:

- `version` – The number following `version` is the version number of the descriptor. The default value is 1.
- `CID` – This line shows the content ID. It is a random 32-bit value updated the first time the content of the virtual disk is modified after the virtual disk is opened. Every link header contains both a content ID and a parent content ID (described below).

  If a link has a parent – as is true of links B and C in the diagram of links in a chain – the parent content ID is the content ID of the parent link.

  If a link has no parent – as is true of link A in the diagram of links in a chain – the parent content ID is set to `ffffffff` (see `parentCID` below).

  The purpose of the content ID is to check the following:

  - In the case of a base disk with a delta link, that the parent link has not changed since the time the delta link was created. If the parent link has changed, the delta link must be invalidated.
  - That the bottom-most link was not modified between the time the virtual machine was suspended and the time it was resumed, or between the time you took a snapshot of the virtual machine and the time you reverted to the snapshot.

- `parentCID` – This line shows the content ID of the parent link – the previous link in the chain – if it exists. If the link does not have any parent (that is, the link is a base disk) the `parentCID` is set to `ffffffff`.
- `createType` – This line describes the type of virtual disk. Flat disk is fully allocated at creation time (pre-allocated). Sparse disk is allocated as needed to store data. Not including legacy types of virtual disk, `createType` can be one of the following:

  - `custom` – descriptor file with arbitrary extents.
  - `monolithicSparse` – single sparse extent with embedded descriptor file.
  - `monolithicFlat` – single flat extent with separate descriptor file.
  - `2GbMaxExtentSparse` – sparse extents 2GB or smaller to account for file system limits.
  - `2GbMaxExtentFlat` – flat extents 2GB or smaller to account for file system limits.
  - `fullDevice` – disk that takes the properties of, and is backed by, physical disk on the host.

- `partitionedDevice` – disk backed by some partitions of physical disk, with other partitions hidden.
- `vmfsPreallocated` – thick (flat) disk on VMFS, with blocks zeroed on first use.
- `vmfsEagerZeroedThick` – pre-allocated (flat) disk on VMFS, with all blocks zeroed when created.
- `vmfsThin` – thin-provisioned VMFS disks that consume only as much space as needed.
- `vmfsSparse` – sparse disk on VMFS, often a redo log, not to be confused with thin-provisioned disk.
- `vmfsRDM` – virtual compatibility raw device map (RDM) acts like a symbolic link to physical disk.
- `vmfsRDMP` – physical compability RDM, similar but sends SCSI commands to underlying hardware.
- `vmfsRaw` – special raw disk for ESXi hosts, passthrough only mode.
- `streamOptimized` – compressed sparse extents with embedded LBA, useful for OVF streaming.

The first seven disk types are for VMware hosted products. Terms that include `monolithic` indicate that the virtual disk is contained in a single file. Terms that include `2GbMaxExtent` indicate that the virtual disk consists of a collection of smaller files. Terms that include `sparse` indicate that a virtual disk starts small and grows to accommodate data. Terms that include `flat` indicate that disk space is allocated at creation time. Product documentation also uses the terms growable and preallocated, respectively.

Terms prefixed by `vmfs` are used for storage on ESXi hosts. TYpe `vmfsSeSparse` is for space-efficient sparse disk used for new-style redo logs. Type `vmfsThick` refers to non-zeroed preallocated disk, and is deprecated. Types `vmfsRawDeviceMap` and `vmfsPassthroughRawDeviceMap` are used in headers for disks that use ESXi raw device mapping.

Types `fullDevice`, `partitionedDevice`, and `vmfsRaw` are used when a virtual machine is configured to make direct use of a physical disk, or partitions on a physical disk, rather than configured to store data in files managed by a host operating system or VMFS.

The term `streamOptimized` is used to describe disks that have been optimized for streaming.

- `parentFileNameHint` – This line, present only if the link is a delta link, contains the path to the parent of the delta link.

The Extents

Each line of the second section describes one extent. The extents are enumerated beginning with the one accessible at offset 0 from the virtual machine's point of view. The format of the line looks like one of the following examples:

```
RW 4192256 SPARSE "test-s001.vmdk"
```

Access | Size in sectors | Type of extent | Filename

```
RW 1048576 FLAT "test-f001.vmdk" 0
```

Access | Size in sectors | Type of extent | Filename | Offset

The extent descriptions provide the following key information:

- Access – may be `RW`, `RDONLY`, or `NOACCESS`
- Size in sectors – a sector is 512 bytes
- Type of extent – may be `FLAT`, `SPARSE`, `ZERO`, `VMFS`, `VMFSSPARSE`, `VMFSRDM`, or `VMFSRAW`.
- Filename – shows the path to the extent (relative to the location of the descriptor).
  If the type of the virtual disk, shown in the header, is fullDevice or partitionedDevice, then the filename should point to an IDE or SCSI block device. If the type of the virtual disk is vmfsRaw, the filename should point to a file in `/vmfs/devices/disks/`.

- Offset – the offset value is specified only for flat extents and corresponds to the offset in the file or device where the guest operating system's data is located. For preallocated virtual disks, this number is zero. For device-backed virtual disks (physical or raw disks), it may be non-zero.

The Disk Database

Additional information about the virtual disk is stored in the disk database section of the descriptor. Each line corresponds to one entry. Each entry is formatted as follows:

```
ddb.<nameOfEntry> = "<value of entry>"
```

When the virtual disk is created, the disk database is populated with entries like those shown in the example descriptor. The entry names are self-explanatory and show the following information:

- The adapter type can be `ide`, `buslogic`, `lsilogic`, or `legacyESX`. The `buslogic` and `lsilogic` values are for SCSI disks and show which virtual SCSI adapter is configured for the virtual machine. The `legacyESX` value is for older ESX/ESXi hosts when the adapter type used in creating the virtual machine is not known.
- The geometry values – for cylinders, heads, and sectors – are initialized with the geometry of the disk, which depends on the adapter type.

There is one descriptor, and thus one disk database, for each link in a chain. Searches for disk database information begin in the descriptor for the bottom link of the chain – Link C in the illustration of links in the chain – and work their way up the chain until the information is found.

Layout of the Example Disk

The link described in the example descriptor has three extents, each of which is a file on disk. The following diagram shows the layout of this link and the filenames of the extents:

Simple Extents

The simplest kinds of extents are backed by a region of a file or a block device. These include the extent types shown in the descriptor as `FLAT`, `VMFS`, `VMFSRDM`, or `VMFSRAW`.

Monolithic or Flat VMDK

A virtual disk described as monolithic and flat consists of two files. One file contains the descriptor. The other file is the extent used to store virtual machine data.

Consider an extent that is described by the following line in a descriptor file.

```
RW 1048576 FLAT "test-f001.vmdk" 0
```

This means that file `test-f001.vmdk` is 1048576 sectors × 512 bytes/sector = 536870912 bytes = 512MB in size.

In VMware ESXi hosts, each link includes only one extent.

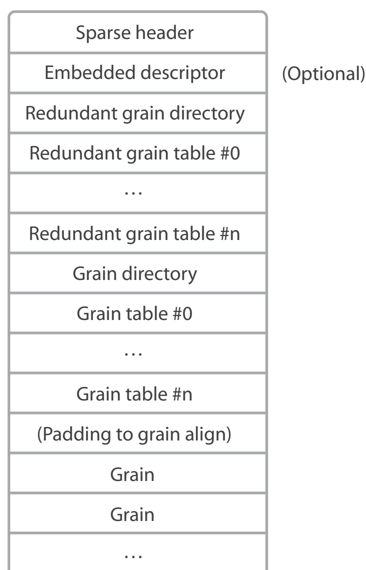
Accessing a Sector in a Flat Extent

Assume you want access to data in a link that is made up of two flat extents. The size of the first extent is C1. The size of the second extent is C2. You want access to sector *x* in the virtual disk, and `x'` is the sector offset in extent 1 or 2 where *x* is located.

- If `x >= C1`, the sector is in extent2. Its relative sector offset is: `x' = x - C1`
- If `x < C1`, the sector is in extent1 at offset x: `x' = x`

Hosted Sparse Extents

In a sparse extent, data storage space is not allocated in advance. Instead, space is allocated as it is needed. A sparse extent also keeps track of whether or not data is represented in the extent. Delta links made up of sparse extents use the copy-on-write semantic. Each sparse extent is made up of the following blocks:



Hosted Sparse Extent Header

The following example shows the content of a sparse extent's header from a VMware hosted product, such as VMware Workstation, VMware Player, VMware Fusion, VMware ACE, or VMware (GSX) Server:

```
typedef uint64 SectorType;
typedef uint8 Bool;
typedef struct SparseExtentHeader {
                  uint32       magicNumber;
                  uint32       version;
                  uint32       flags;
                  SectorType   capacity;
                  SectorType   grainSize;
                  SectorType   descriptorOffset;
                  SectorType   descriptorSize;
                  uint32       numGTEsPerGT;
                  SectorType   rgdOffset;
                  SectorType   gdOffset;
                  SectorType   overHead;
                  Bool         uncleanShutdown;
                  char         singleEndLineChar;
                  char         nonEndLineChar;
                  char         doubleEndLineChar1;
                  char         doubleEndLineChar2;
                  uint16       compressAlgorithm;
                  uint8        pad[433];
               } SparseExtentHeader;
```

This structure needs to be packed. If you use `gcc` to compile your application, you must use the keyword `__attribute__((__packed__))`.

Notes:

- All the quantities defined as `SectorType` are in sector units.
- `magicNumber` is initialized with
  `#define SPARSE_MAGICNUMBER 0x564d444b /* 'V' 'M' 'D' 'K' */`
  This magic number is used to verify the validity of each sparse extent when the extent is opened.

- `version` – The version number can be 1 or 2. See “Version 2 Hosted Sparse Extents” on page 7.

  `SparseExtentHeader` is stored on disk in little-endian byte order, so if you examine the first eight bytes of a VMDK file, you see ‘K’ ‘D’ ‘M’ ‘V’ `0x01 0x00 0x00 0x00` or ‘K’ ‘D’ ‘M’ ‘V’ `0x02 0x00 0x00 0x00`.

- `flags` contains the following bits of information in the current version of the sparse format:
  - bit 0: valid new line detection test.
  - bit 1: redundant grain table will be used.
  - bit 2: zeroed-grain GTE will be used. See “Version 2 Hosted Sparse Extents,” below.
  - bit 16: the grains are compressed. The type of compression is described by `compressAlgorithm`.
  - bit 17: there are markers in the virtual disk to identify every block of metadata or data and the markers for the virtual machine data contain logical block addressing (LBA).
- `grainSize` is the size of a grain in sectors. It must be a power of 2 and must be greater than 8 (4KB).
- `capacity` is the capacity of this extent in sectors – should be a multiple of the grain size.
- `descriptorOffset` is the offset of the embedded descriptor in the extent. It is expressed in sectors. If the descriptor is not embedded, all the extents in the link have the descriptor offset field set to 0.
- `descriptorSize` is valid only if `descriptorOffset` is non-zero. It is expressed in sectors.
- `numGTEsPerGT` is the number of entries in a grain table. The value of this entry for virtual disks is 512.
- `rgdOffset` points to the redundant level 0 of metadata. It is expressed in sectors.
- `gdOffset` points to the level 0 of metadata. It is expressed in sectors.
- `overHead` is the number of sectors occupied by the metadata.
- `uncleanShutdown` is set to FALSE when VMware software closes an extent. After an extent has been opened, software checks for the value of uncleanShutdown. If TRUE, the disk is checked for consistency and uncleanShutdown is set to TRUE after this consistency check. Thus, if the software crashes before the extent is closed, this boolean is found to be set to TRUE the next time the virtual machine is powered on.
- Four entries are used to detect when an extent file has been corrupted by transferring it using FTP in text mode. The entries should be initialized with the following values:

```
singleEndLineChar = '\n';
nonEndLineChar = ' ';
doubleEndLineChar1 = '\r';
doubleEndLineChar2 = '\n';
```

- `compressAlgorithm` designates the algorithm to compress every grain in the virtual disk. If bit 16 of the `flags` field is not set, `COMPRESSION_NONE` is assumed. The deflate algorithm is described in RFC 1951.

```
#define COMPRESSION_NONE    0
#define COMPRESSION_DEFLATE 1
```

Version 2 Hosted Sparse Extents

Recent VMware hosted platform products support a new “zeroed-grain” grain table entry (GTE). The zeroed-grain GTE returns all zeros on read. In other words, the zeroed-grain GTE indicates that a grain in the child disk is zero-filled but does not actually occupy space in storage. A sparse extent with zeroed-grain GTE has the following in its header:

- `SparseExtentHeader.version` = 2
- `SparseExtentHeader.flags` has bit 2 set

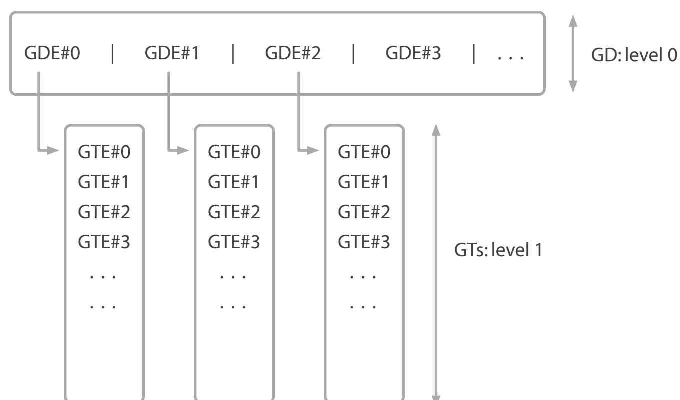
Other than the new flag and the possibly zeroed-grain GTE, version 2 sparse extents are identical to version 1. Also, a zeroed-grain GTE has value `0x1` in the GT table (for details, see “Summary” on page 9). Currently version 2 hosted sparse extents occur when you shrink a child disk (also called snapshot). They may occur in other circumstances. When a shrink operation (also called compact) is done on a version 1 child disk, the version number is upgraded to 2, and the compacted disk takes up less space than it would otherwise.

Releases before Workstation 5 cannot read version 2 sparse disks, but all releases of VMware Fusion can.

Products may (but are not required to) downgrade a version 2 sparse extent to version 1 if the extent no longer contains a zeroed-grain GTE. This is done by setting `version = 1` and setting bit 2 of `flags` to 0.

Hosted Sparse Extent Metadata

There are two levels of metadata in a sparse extent from a hosted VMware product. Level-0 metadata is called a grain directory or a GD. Level-1 metadata is called a grain table or a GT. Each entry in the level-0 metadata points to a block of level-1 metadata, as shown in the following diagram:



Redundancy

VMware software keeps two copies of the grain directories and grain tables on disk to improve the virtual disk's resilience to host drive corruption.

Grain Directory

Each entry in a grain directory is called a grain directory entry or GDE. A grain directory entry is the offset in sectors of a grain table in a sparse extent. The number of grain directory entries per grain directory (the size of the grain directory) depends on the length of the extent. A grain directory entry is a 32-bit quantity.

Grain Table

Each entry in a grain table is called a grain table entry or GTE. A grain table entry points to the offset of a grain in the sparse extent. There are always 512 entries in a grain table, and a grain table entry is a 32-bit quantity. Consequently, each grain table is 2KB.

In a newly created sparse extent, all the grain table entries are initialized to 0, meaning that the grain to which each grain table entry points is not yet allocated. Once a grain is created, the corresponding grain table entry is initialized with the offset of the grain in the sparse extent in sectors.

All the grain tables are created when the sparse extent is created, hence the grain directory is technically not necessary but has been kept for legacy reasons. If you disregard the abstraction provided by the grain directory, you can redefine grain tables as blocks of grain table entries of arbitrary size. If there were no grain directories, there would be no need to impose a length of 512 entries.

Grain

A grain is a block of sectors containing data for the virtual disk. The granularity is the size of a grain in sectors. It is a property of the extent and is specified in the sparse extent header as `grainSize`. The default is currently 128, thus each grain contains 64KB of virtual machine data. The size of a sparse extent should be a multiple of `grainSize`. Each grain starts at an offset that is a multiple of the grain size.

Accessing a Sector in a Hosted Sparse Extent

Assume you want access to data in sector *x* stored in a link containing a single sparse extent. You need to locate the grain containing this sector (if it exists) by first looking up the grain directory entry to find the location of the grain table that records the grain's location.

If grainSize is defined as

grain = 2^G sectors

then the area accessible with a single grain table is

$$\begin{aligned} gtCoverage &= \text{number of GTEs per GT} \times grainSize \\ &= 512 \times 2^G \\ &= 2^9 \times 2^G \\ &= 2^{9+G} \text{ sectors} \end{aligned}$$

If the grainSize is 128 sectors, then:

$$\begin{aligned} gtCoverage &= 2^{9+7} \\ &= 2^{16} \text{ sectors} \\ &= 32MB \end{aligned}$$

To verify that the grain containing the sector has been allocated, you must examine a grain table. To find the grain table you need, examine the grain directory entry at offset floor(*x*/gtCoverage) in the grain directory.

```
GDE = GD [ floor(x/gtCoverage) ]
```

Function floor is defined as: `floor(s)` is an integer such that

```
floor(s) ≤ s < floor(s) + 1
```

Using this grain directory entry, you can locate the grain table. The grain you want is pointed to by

```
GTE = GT [ floor( (x % gtCoverage) / grainSize) ]
```

If GTE is 0, it means the grain is not yet allocated. All the reads in this grain return sectors of 0s (unless there is a parent link). The first write allocates a grain. If there is no parent, the grain is initialized with 0s. If there is a parent link, you need to respect the copy-on-write semantic and initialize the content of the grain by reading from the parent.

If GTE is 1, that means the grain is all zeros. All the reads in this grain return sectors of zeros, even if there is a parent link. The first write allocates a grain, which is initialized with zeros.

Summary

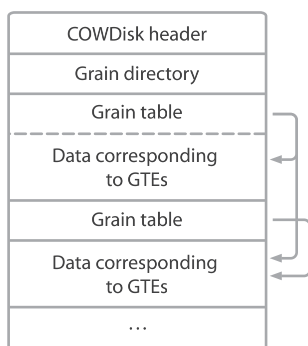
- `GDE = GD [ floor(x / 2^(9+G)) ]`
- `GTE = GT [ floor((x % 2^(9+G)) / 2^G) ]`
- `[ GTE == 0 ]` <==> [grain is not present, thus
  reads with no parent: return 0s;
  reads with a parent: read from parent;
  writes: allocate a grain and write to it]
- `[ GTE == 1 ]` <==> [grain is zero – reads: return zeros; writes: allocate a zeroed grain and write to it]
- `[ GTE > 1 ]` <==> [grain is present, read from and write to it]

ESXi Host Sparse Extents

Sparse extents in ESXi hosts have a different layout from those in the hosted products. The sparse extent header in an ESXi host refers to the sparse extent as a copy-on-write (COW) disk. There are two levels of metadata in a sparse extent on an ESXi host.

- The first level, or the grain directory, refers to the set of grain directory entries (GDEs), where each GDE covers COW_NUM_LEAF_ENTRIES (=4096) * granularity (=512 bytes) = 2MB of data. The grain directory is stored after the COWDisk header and is updated when a new GDE is initialized or modified.
- The second level in the copy-on-write metadata is a grain table (GT). The grain table is 16KB in size and covers 4096 data sectors. A new GT is allocated when a new GDE is added and is modified when a new GTE is allocated.

A GT is followed by the data sectors corresponding to its GTEs. Because delta links (also called redo logs) are sparse, all the data sectors are not allocated immediately after a GT. The following diagram shows the layout:



ESXi Host Sparse Extent Header

The following example shows the content of a sparse extent's header on an ESXi host:

```
#define COWDISK_MAX_PARENT_FILELEN  1024
#define COWDISK_MAX_NAME_LEN        60
#define COWDISK_MAX_DESC_LEN        512
typedef struct COWDisk_Header {
                   uint32         magicNumber;
                   uint32         version;
                   uint32         flags;
                   uint32         numSectors;
                   uint32         grainSize;
                   uint32         gdOffset;
                   uint32         numGDEntries;
                   uint32         freeSector;
                   union {
                       struct {
                           uint32   cylinders;
                           uint32   heads;
                           uint32   sectors;
                       } root;
                       struct {
                           char     parentFileName[COWDISK_MAX_PARENT_FILELEN];
                           uint32   parentGeneration;
                       } child;
                   } u;
                   uint32         generation;
                   char           name[COWDISK_MAX_NAME_LEN];
                   char           description[COWDISK_MAX_DESC_LEN];
                   uint32         savedGeneration;
                   char           reserved[8];
                   uint32         uncleanShutdown;
                   char           padding[396];
               } COWDisk_Header;
```

Notes:

- `magicNumber` is set to `0x44574f43` which is ASCII `COWD`.
- `version` – The value of this entry should be 1.
- `flags` is set to 3.
- `numSectors` refers to total number of sectors on the base disk.
- `grainSize` is the granularity of data stored in delta links. This varies from one sector (the default) to 1MB.
- `gdOffset` starts at the fourth sector, because the `COWDisk_Header` structure takes four sectors.
- `numGDEntries` is `CEILING(numSectors, gtCoverage)`

- `freeSector` is the next free data sector. It must be less than the length of the delta link. It is initially set to `gdOffset + numGDSectors;`
- `savedGeneration` is used to detect the unclean shutdown of the delta link. It is initially set to `0`.
- `uncleanShutDown` is used to trigger the metadata consistency check in case there is an abnormal termination of the program.
- The remaining fields are not used. They are present for compatibility with legacy virtual disk formats.

ESXi Host Sparse Extent Metadata

The metadata for an ESXi host sparse extent is similar to that for a sparse extent in a hosted VMware product, as described in "Hosted Sparse Extent Metadata" on page 8, with the following exceptions:

- ESXi sparse extents do not include redundant copies of the grain directory.
- Grain tables have 4096 entries.
- Each grain contains 512 bytes.

Accessing a Sector in an ESXi Host Sparse Extent

The method for accessing a sector in an ESXi host sparse extent is similar to that described in "Accessing a Sector in a Hosted Sparse Extent" on page 8. Be sure to allow for the differences in metadata described above.

Stream-Optimized Compressed

Stream-optimized compressed extents are meant to be easily streamed over a network link. They are designed to minimize the memory footprint of the server streaming the virtual disk and also allow for the use of a simple client application to read the virtual disk data. This virtual disk type is used primarily in the monolithic form, typically for delivery of OVF virtual appliances.

Each stream-optimized compressed sparse extent is made of the following blocks:

Sparse header
Embedded descriptor
Grain marker
Compressed grain
...
Grain table marker
Grain table
Grain marker
Compressed grain
...
Grain table marker
Grain table
[...]
Grain directory marker
Grain directory
Footer marker
Footer
End-of-stream marker

Each marker and its associated block begin on a sector or 512-byte boundary. Each marker can be seen as a C structure with the following layout:

```
struct Marker {
                    SectorType val;
                    uint32     size;
                    union {
                       uint32  type;
                       uint8   data[0];
                    } u;
               };
```

There are five types of markers: compressed grain markers, grain table markers, grain directory markers, footer markets, and end-of-stream markers. Grain markers are indicated by a non-zero size so there is no type ID for them.

```
#define MARKER_EOS      0
#define MARKER_GT       1
#define MARKER_GD       2
#define MARKER_FOOTER   3
```

Based on the values of `val`, `size`, and `type`, you can distinguish between the various types of markers and their associated blocks. Additional types may be defined in the future to indicate various metadata elements.

In the following discussion of marker types, `m` is a pointer to a marker defined by the `Marker` structure.

Compressed Grain Marker

Pointer `m` is a marker for a compressed grain if `m->size != 0`. In this case, the marker and block have the following layout:

```
struct GrainMarker {
                    SectorType lba;
                    uint32     size;
                    uint8      data[0];
               };
```

In this structure:

- `lba` is the offset in the virtual disk where the block of compressed data is located
- `size` is the size of the compressed data in bytes
- `data` is the data compressed with RFC 1951

End-of-Stream Marker

Pointer `m` is an end-of-stream marker if `m->size == 0 && m->u.type == MARKER_EOS`. The end-of-stream marker signals the end of the virtual disk. Each end-of-stream marker is padded to occupy a sector. The structure looks like this:

```
struct EOSMarker {
                    SectorType val;
                    uint32     size;
                    uint32     type;
                    uint8      pad[496];
               };
```

In this structure:

- `val` is `0`.
- `size` is `0`.
- `type` is `MARKER_EOS (0)`.
- `pad` is unused. It must be written as zero and ignored on read.

Metadata Markers

Markers used to signal the blocks containing grain tables, grain directories, or footers have the same layout.

If `m->size == 0 && m->u.type == MARKER_GT`, `m` is a marker for a grain table.

If `m->size == 0 && m->u.type == MARKER_GD`, `m` is a marker for a grain directory.

If `m->size == 0 && m->u.type == MARKER_FOOTER`, `m` is a marker for a footer.

These markers and the blocks of data they signal have the following layout:

```
struct MetaDataMarker {
                     SectorType numSectors;
                     uint32     size;
                     uint32     type;
                     uint8      pad[496];
                     uint8      metadata[0];
                };
```

In this structure:

- `numSectors` is the number of sectors occupied by the metadata, excluding the marker itself.
- `size` is `0`.
- `type` is one of `MARKER_GT (1)`, `MARKER_GD (2)`, or `MARKER_FOOTER (3)`.
- `pad` is unused. It must be written as zero and ignored on read.
- `metadata` points to a grain table if type is `MARKER_GT`, a grain directory if type is `MARKER_GD`, or a footer if type is `MARKER_FOOTER`.

Header and Footer

The header and the footer are both described by the same `SparseExtentHeader` structure shown in "Hosted Sparse Extent Header" on page 6. The footer takes precedence on the header when it exists. The footer should be the last block of the disk and immediately followed by the end-of-stream marker so that they together occupy the last two sectors of the disk.

Stream-optimized compressed sparse disks differ from regular sparse disks in that:

- `flags` has bits 16 and 17 set to indicate that the grains are compressed and that each block of metadata or data is identified by a marker.
- `compressAlgorithm` is set to `COMPRESSION_DEFLATE (1)`.

  This compression algorithm is described in RFC 1951.
- The `rgdOffset` should be ignored because bit 1 of the `flags` field is not set.

The header and footer differ in that the field `gdOffset` is set to

```
#define GD_AT_END    0xffffffffffffffff
```

in the copy of the header stored at the very beginning of the extent, whereas it is set to the proper value for the copy of the header (footer) that is stored at the end of the extent.

Glossary

Chain – A collection of disk links that can be accessed as a single entity.

Child disk – A disk link in a disk chain that has a parent link.

Delta link – A link made of one or more sparse extents. It is a difference link, a child of a parent link. It contains only data that the guest operating system has written to the disk after the creation of the delta link. It allows software to go back in time and, by simply removing the delta link, restore the content of the disk to its state immediately before creation of the delta link. Delta links are also called redo-log files.

Descriptor – Data about the disk abstraction, such as total space or an extent list. The descriptor may be in a separate file or embedded in the header of a sparse extent. An embedded disk descriptor is placed in the first extent of a disk link rather than in a separate disk descriptor file. An embedded disk descriptor can be used only when the first extent of a link is sparse.

Disk – A disk chain that appears to the guest operating system as a single physical disk.

Disk database – A name-value pair text database found in the disk descriptor. It contains information that the disk library does not need for disk function. Examples of these kinds of values are virtual hardware version and VMware Tools version.

Extent – A region of a disk link backed by a region of a file or device. An extent can be sparse, flat, or device. An extent does not have notions of disk properties but acts purely as storage of a certain size. A flat extent is an extent backed by a flat file. Flat extents are also called plain or preallocated. A sparse extent is an extent that does not allocate its data storage space in advance, but allocates as it goes along, and keeps track of whether or not data is represented in the extent. Sparse extents are also called growable.

Flat – Space in a VMDK is fully allocated at creation time (pre-allocated). Contrast with sparse.

Grain – A block of sectors containing data for the virtual machine's disk. Granularity defines the size of a grain. Each grain table entry points to one grain.

Granularity – The size of a single grain in a sparse extent.

Grain directory – Metadata identifying the locations of grain tables. The grain directory is ignored by recent VMware programs because the grain table is allocated in advance.

Grain table – Metadata identifying the locations of grains.

Link – A single node in a disk chain. A link consists of one or more extents.

Parent link – A link that has a child. A parent may itself have a parent.

Sparse – Space in a VMDK is allocated only when needed to store data (growable). Contrast with flat.

If you have comments about this documentation, submit your feedback to: docfeedback@vmware.com

VMware, Inc. 3401 Hillview Ave., Palo Alto, CA 94304 www.vmware.com

Copyright © 2007, 2011 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents. VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

Item: EN-000777-00
Updated: 12/20/11