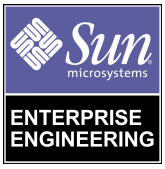
VxVM Private Regions: Mechanics & Internals of the VxVM Configuration Database

---

By Gene Trantham - Enterprise Engineering

Sun BluePrints™ OnLine - July 2000



http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 806-6194-10
Revision 01, July 2000

Please
Recycle

VxVM Private Regions: Mechanics & Internals of the VxVM Configuration Database

Introduction

Veritas Volume Manager (VxVM) is a storage management software package that is popular on large scale Solaris™ Operating Environment systems. However, VxVM is often poorly understood by System Administrators and this lack of understanding leads to poor configurations that could hinder performance or system recovery. Central to VxVM's storage management, is the VxVM configuration database, also known as the private region. Understanding the VxVM private region will provide insight into how VxVM functions and also gives System Administrators the ability to configure VxVM, rather than circumvent it.

VxVM enables the System Administrator to manipulate the physical media attached to a host into virtual devices that are more effective than straight media. These virtual devices are configured with various RAID characteristics (mirror, stripe, RAID_5, etc) that are not available with physical devices themselves. VxVM must track the details of its device configuration quite closely, yet it is not permitted to store that information on any of the virtual devices it creates. To do so would place VxVM into a chicken-egg deadlock at system startup time.

VxVM stores its configuration data in an internal relational database. Fundamentally, this database operates like most generic databases. Data is retrieved and changed via a special interface, and referential integrity is maintained according to strict internal rules. More importantly, VxVM's database stores its data using a proprietary format on "raw" partitions. These partitions are scattered among the very disks that VxVM is charged with managing, yet are distinct from the space on these disks which is to hold the virtual devices themselves.

To accommodate this split, VxVM divides each disk it is to manage into two distinct regions: *public* and *private*. The *public region* comprises the bulk of the space on the disk device. It is used to create the virtual devices upon which the system's data is stored: file systems, swap devices, database data, etc. The *private region* is used by VxVM as its internal storage for configuration information and virtual device definitions. This space cannot be used to create virtual devices, and must remain dedicated for VxVM's exclusive internal use.

Private regions are a critical part of the VxVM storage management system, however, they are not able to take advantage of that management system because they contain metadata crucial to VxVM's startup. Therefore, the private regions must be manipulated by low-level VxVM utilities, and they must be created and accessed according to the low-level rules of physical media and standard partitioning. This is in part due to the criticality of the private region, and partly to hide the unsightly details of its management from the user. VxVM controls the private regions with minimal input from the user and minimal feedback about its actions. The VxVM user can virtually ignore the fact that private regions even exist.

Rules for a Peaceful Co-Existence with VxVM

Unfortunately, VxVM cannot hide private regions completely, nor can it completely eliminate the work necessary to create and manipulate them. The key to understanding the internal workings of VxVM and how it handles media is to understand how it manipulates the private region. To comprehend how VxVM operates internally, it is important to know some of the rules that apply. The following list explains the basic rules and why VxVM must follow them.

RULE #1: Do Not Write to Block Zero

The first data block on any partitioned disk device holds the Volume Table of Contents (VTOC). This table defines the boundaries of each partition/slice on the disk. This block is exceptionally important to the life of the disk, yet is not protected in any way by the disk itself. Block zero can be written with arbitrary data by any process that has write authority on the device. Almost all data layouts in the Solaris Operating Environment account for the VTOC block and simply skip over it before writing their own data. A UFS file system, for example, does not write any data into the first 512 bytes on a partition. A similar offset is used by other file system types, and even the VxVM private region.

The VTOC controls and reports what slices are mapped to what space on the device. See `/usr/include/sys/vtoc.h` and `/usr/include/sys/dklabel.h` for a complete description of the fields and formats stored here. If the VTOC is ever corrupted or erased, the device driver will not be able to ascertain the correct offset and length of the various partitions defined on the disk. This implies that it will be unable to fetch the correct data blocks when they are requested. Or, even worse, it will silently fetch the wrong data blocks on reads and store data in the wrong locations on writes. There is no way to know if the data is valid in this case.

RULE #2: Only 8 Slices per disk

The Solaris Operating Environment VTOC permits only 8 slices/partitions to be defined on a device. This 8 slice limit is set at kernel compile time to either 8 or 16 slices, depending on the platform. SPARC™ systems are always compiled for 8 slices (see `/usr/include/sys/vtoc.h`, `/usr/include/sys/dklabel.h`, and `/usr/include/sys/isa_defs.h`).

RULE#3: Slices Can Overlap

The definition of a slice within the VTOC is merely a start cylinder and the length of the slice in cylinders. A given space on the disk does not know to which slice(s) it belongs. There is no restriction that prohibits two slices from mapping some or all of the same physical space on the disk. For example, the traditional backup slice, slice two, maps the entire disk space; yet, this does not prevent other slices from mapping subsets of this space.

RULE#4: All Slices Begin and End on a Cylinder Boundary

All slices inthe Solaris Operating Environment begin and end on cylinder boundaries, from the disk point of view there is no technical reason for this but this requirement remains mostly for historical reasons. Slices are defined as a start cylinder and a length in cylinders.

With advances in disk technology over the past several years, the meaning of the cylinder boundary has changed. The internal geometry of disk drives, sectors, tracks, and cylinders are no longer fixed. In the past, a fixed geometry drive contained a number of cylinders that all contained exactly the same number of sectors, however, this turned out to be very inefficient in terms of space usage. Most drives today are Zone Bit Rate (ZBR) drives.

A zone is a collection of relatively few tracks very close together on the platter. For any given zone on the platter surface, there are the same number of sectors. As you get farther out on the platter surface, more sectors can be squeezed onto those tracks and the linear velocity of the platter surface increases as the bits go under the head at a faster rate. Consequently, for the zone closest to the spindle, the bit rate is fairly low, and the number of sectors per track is low. On the zone closest to the rim of the platter, the bit rate is at its highest and there are more sectors per track.

For example, an average SCSI disk:

```
# prtvtoc /dev/rdsk/c3t0d0s2
* /dev/rdsk/c3t0d0s2 partition map
*
* Dimensions:
*     512 bytes/sector
*     135 sectors/track
*      16 tracks/cylinder
*    2160 sectors/cylinder
*    4392 cylinders
*    3880 accessible cylinders
*
   [ ... some output deleted ... ]
```

This is a ZBR drive, yet it reports that it has a fixed geometry of 2160 sectors per cylinder. Additionally, there are not 135 sectors for each track on that disk. The technical specifications from this drive's manufacturer indicate that the average number of sectors per track is 114. The reason for this misinformation is due to the fact that most of the device driver code handles disk drives as if the disk is of fixed geometry. The auto-detect feature of the drive simply lies about its geometry to prevent the device driver from attempting to calculate the zone sizes.

A similar lie is told regarding the number of cylinders accessible in this disk. According to `prtvtoc`, there are 4392 cylinders on this drive, but only 3880 of them are accessible. What happened to the other 512 cylinders? There are actually 4392 cylinders on this drive, however, because the disk lied about how big each cylinder is, it must also lie about how many of these cylinders are accessible in order to make the total drive size add up.

The 3880 cylinders that are labeled accessible are actually virtual cylinders. Each of the virtual cylinders have 135 virtual sectors each. In fact, some physical cylinders have fewer than that, some more. On the inner tracks, it may take 2 or 3 physical cylinders to make up a virtual cylinder. On the outer tracks, there may be 2 or more virtual cylinders per physical cylinder.

The number of heads (tracks/cylinder) on this drive is claimed to be 16. This is probably true, but there is no way to know from this output. The technical specifications from the drive vendor indicate that there are actually 19 heads.

It is important to note that Sun uses custom disk drive firmware to ensure that the virtual cylinder information is the same for all disks of like size. This allows Sun to use multiple suppliers of disk drive technology without causing problems with software or firmware that expects all disks in a subsystem to have the same virtual configuration.

Disk Types & Private Region Mechanics

Disks under volume manager control must be readily identifiable as such to the volume manager. The private region is a mechanism to do this. All disks under VxVM management are labeled with a stamp or header recognizable to the VxVM configuration daemon. There are a number of ways that this header/stamp space can be handled. Disk devices are categorized into three basic types for VxVM purposes based upon how it defines and tracks the private region. Disk types are:

- "nopriv" No private region defined; public region (data) only.
- "sliced" A disk which supports two or more slices/partitions. The public region is on one partition, the private region on another partition.
- "simple" A disk which has no partitioning available. The private and public regions are defined on the same logical device.

"NOPRIV disks"

This is a special case. It is only useful for transient devices such as `ramdisks` implemented in system memory. A `nopriv` disk must exist in disk groups which have simple or sliced disks defined. The group information, since it cannot be located on the `nopriv` disk, must be available on other disks within the group. `nopriv` disks also cannot change their hardware location or address across reboots; VxVM has no way of self-identification on a `nopriv` disk.

"SLICED disks"

A sliced disk is the default type. Sliced disks are those which support partitioning on the LUN. Sliced disks are the default because they are the most common in Solaris (all SCSI disks support partitioning), and the public/private distinction is much easier to work with. The public region is a defined partition, as is the private region. They are separately addressable, therefore offset/length computations are not necessary within VxVM.

"SIMPLE disks"

Simple disks are those which do not support partitioning. The distinction between public and private region on such a disk is handled by VxVM internally using offsets. Data space (public region) begins at some offset into the LUN's address space. That offset is defined by VxVM, and is retrievable using '`vxdisk list`' on the target disk.

```
# vxdisk init c1t3d3s1 type=simple
# vxdg -g rootdg adddisk TestDisk=c1t3d3s1
# vxdisk -g rootdg list | grep simple
c1t3d3s1     simple    TestDisk     rootdg       online
# vxdisk list TestDisk
Device:    c1t3d3s1
devicetag: c1t3d3
type:      simple
hostid:    rtfm
disk:      name=TestDisk id=943977664.1229.rtfm
group:     name=rootdg id=943917799.1025.rtfm
info:      privoffset=1
flags:     online ready autoimport imported
pubpaths:  block=/dev/vx/dmp/c1t3d3s1 char=/dev/vx/rdmp/c1t3d3s1
version:   2.1
iosize:    min=512 (bytes) max=2048 (blocks)
public:    slice=1 offset=1025 len=2096575
   [ ... ]
```

Note that the public region is defined on slice 1, with an offset of 1025 sectors. The private region is also in slice 1, with an offset of 1 sector and a length of 1024. No slice boundary separates public from private space on this simple device, meaning that VxVM must compute the offsets (as declared above) each time it needs to access any bit of data within this device. Also note that VxVM is aware of the VTOC in block 0 and skips over that block to begin writing at block 1.

So why have a 'simple' type? All practical, modern disk drives support partitioning of some sort. It is occasionally useful to partition a drive by hand, then treat each of these separate partitions as a SIMPLE disk. The custom defined partitions may not be further partitioned, so they function as if they were unpartitionable disks.

Two cases where this is useful:

1. Creating a stub `rootdg` out of a 2-cylinder slice on a disk.

   On occasion, it is useful to have no `rootdg`. Because volume manager must have a `rootdg` in order to function, it is good practice to make it as small and unobtrusive as possible. The easiest way to do this is to create a small slice (about 2 cylinders), define that slice as a simple disk to the volume manager, and place that simple disk into `rootdg`.

   If you do this, you will preclude the use of VxVM to manage your boot device. The root file system must be on a volume called '`rootvol`' in '`rootdg`' or VxVM will not boot from it.

2. Cutting a disk into multiple slices so each partition can be used as a separate disk media object.

   In some obscure cases, it is useful to split very large spindles into smaller virtual disks (defined as simple disks to VxVM) to ease management of the space. Suppose your data profile needed 4 disk groups, but only needed 2 Gb of mirrored space in each group. This would be ideal for a rack full of 2Gb disks, but suppose your only option was 18Gb spindles. It would be more effective to split each one into 4 simple disks each, and implement all four disk groups on two spindles. However, there are certainly drawbacks to doing this.

Sliced disks are by far the most common and the most useful. `nopriv` and Simple disk types serve specific roles in very special important cases, but are infrequently encountered. It is much more important to know how a Sliced disk behaves.

Sliced disks become known to the volume manager during the VxVM disk scan. The VxVM configuration daemon examines all disk devices it can find in an attempt to locate a partition which is TAGGED with the special VxVM-Private-Region tag. Public regions have a similar tag. (Recall that tags are a 4-bit value saved in the VTOC and used to identify the purpose of a partition).

The VxVM tag values are defined in `vxvm/vollocal.h` (part of the `VRTSvmdev` package). As of this writing, the public region tag is 14 (0xe) for all versions of VxVM. The private region is tagged with 15 (0xf). This tag is what defines a space as being the private region. Do not assume that the contents or the header information stamped into the first few sectors, define a space as being the private region.

The following example demonstrates what happens when a partition tag is changed. Take any unused disk, initialize that disk and add it to a disk group. A `prtvtoc` will show the size, location, and tags associated with each

```
# /etc/vx/bin/vxdisksetup -i c1t3d3
# vxdg -g rootdg adddisk TestDisk=c1t3d3
# prtvtoc /dev/rdsk/c1t3d3s2
  [ . . . ]
*                          First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount
Directory
       2      5    01          0   4154160   4154159
       3     15    01          0      1520      1519
       4     14    01       1520   4152640   4154159
# vxdisk -g rootdg list
DEVICE       TYPE      DISK         GROUP        STATUS
c0t1d0s2     sliced    rootdisk     rootdg       online
c0t3d0s2     sliced    rootmirror   rootdg       online
c1t3d3s2     sliced    TestDisk     rootdg       online
```

Slice 2 is maintained as the backup slice which maps the entire address space of the physical media. Slices 3 and 4 are the private and public regions, respectively. Note the tag of 15 on the private region. Also note that it is exactly one cylinder long. The public region maps the remaining space on the device.

To demonstrate these concepts, consider the following commands which remove the public and private region tags, yet they do not alter the sizes, locations, or contents of the VxVM-defined slices (slice 3 and 4). After changing the tags, the following commands force VxVM to re-scan for disks in order to see any effects from this change.

```
# fmthard -d 3:0:0:0:1520 /dev/rdsk/c1t3d3s2
# fmthard -d 4:0:0:1520:4152640 /dev/rdsk/c1t3d3s2
# prtvtoc /dev/rdsk/c1t3d3s2
  [ . . . ]
*                          First     Sector    Last
* Partition  Tag  Flags    Sector     Count    Sector  Mount
Directory
       2      5    01          0   4154160   4154159
       3      0    00          0      1520      1519
       4      0    00       1520   4152640   4154159
# vxconfigd -k -m enable
# vxdisk -g rootdg list
DEVICE       TYPE      DISK         GROUP        STATUS
c0t1d0s2     sliced    rootdisk     rootdg       online
c0t3d0s2     sliced    rootmirror   rootdg       online
-            -         TestDisk     rootdg       failed
was:c1t3d3s2
```

Notice that VxVM was unable to locate a disk with `TestDisk`'s private region. In the absence of a disk which answers to that name, it must assume that it has failed or has been removed from the system. `TestDisk` is failed and its last reported whereabouts are reported.

Note again that the only change made using `fmthard` was the tag assigned to each slice. The start and length of each remained the same. No changes were made to the contents of the private region at all.

This allows Sliced disks to be partitioned into two broad regions using the device driver's underlying ability to partition its devices. One of the partitions is declared the private region and it is marked as such using the VxVM magic value of 15. The remaining space on the device is mapped into the public region partition, and tagged with the magic value 14. Tags and flags are the only characteristic that the configuration daemon uses to discern the character of a slice or its contents.

Private Region Location

The private region is defined as a slice entry and therefore is subject to the limitations imposed on slices. Furthermore, it must be contiguous, aligned on cylinder boundaries, and it is defined in terms of a start address (sector) and a length. No restriction is made within VxVM as to where the private region must be on the disk; only that it be a slice tagged with 15 in the type field.

1. Private region at the beginning

Block zero of a sliced disk is used to hold the VTOC of the physical device. Nothing should attempt to write to offset zero on any such disk, or it will place the contents of the VTOC in jeopardy. File systems and boot blocks all offset at least one sector when writing to a slice for exactly this reason. Private regions for VxVM also do this to preserve the VTOC contents on that block.

A private region is placed at the beginning of a disk by default when that disk is initialized. Block zero is protected by the fact that the data layout for a private region does not place anything in location zero. The private region can also be placed at the beginning of a disk during its encapsulation. The circumstances which permit this are exceedingly rare, and typically only arise from careful manipulation of the data ahead of time.

2. Private region at the end

A private region can be placed at the end of the disk if it is initialized with the '`-e`' option to `vxdisksetup`. In so doing, no protection is granted to the critical VTOC in block 0 so all subdisk offsets on this disk must be computed at N+1. It works, but it is kind of a hack and can be confusing when you are trying to map to slices later.

An encapsulated disk is a disk with existing data to be preserved, such as the boot disk, that is brought under Volume Manager control. In the case when a disk is encapsulated, yet does not map all of its disk space to slices, the private regions are often placed at the end of the disk. If free space exists at the end, that is where the private region will be placed during the encapsulation process. Block zero is 'protected' in such cases by mapping a single-block subdisk at offset zero. That subdisk is marked in such a way that it is not movable using conventional means.

3. Private region somewhere in the middle

The private region of a VxVM may even be at some arbitrary location within the address space of the disk. This is the case when a disk is encapsulated, and the only 'free' space in which to place the private region is in the center of the disk. The most common example of this is a boot disk encapsulation.

When a boot device is encapsulated, space for the private region is very typically "stolen" from the last few cylinders of the swap space. Swap is shortened by the required amount, and the private region slice is mapped into the now-available diskspace. Since swap is almost never at the end of a disk, this places the new private region somewhere in the middle of the disk.

This causes a number of problems. The most difficult is how to represent a single private and a single public region on the disk. Recall that each region must be a slice, and that slices are defined in terms of a starting point and offset. In the geometry of our private-region-in-the-middle disk, we have some amount of data/public space, a private region, then more data/public space. The private region is easy to map to a slice. But what about the public region? It is non-contiguous; the private region separates the two public data spaces. One way to solve this problem is to overlap the two regions.

The public region is mapped to the entire disk, just as the backup slice (s2) is. The private region is mapped to a slice as normal. This means that the private region lies within the address space of both the public and private region's partitions. In order to prevent a data volume from being created out of the space occupied by the private region, VxVM must mask off that space in some way. It does this by creating a subdisk within the public region which corresponds to the private region's location.These placement methods are necessary for dealing with encapsulated disks in general and specifically with the boot disk. In most systems, VxVM manages the boot device by encapsulating it. That disk has typically just been installed with the Solaris operating system, and every bit of available space on the device has been devoted to one file system or another in the standard OS install (or swap). Prior to release of the Solaris 8 Operating Environment, this almost always meant two things: the root file system begins at cylinder zero on the disk and the primary swap partition is somewhere in the middle of the disk. These two conditions gave rise to two special subdisks in `rootdg`: `rootdisk-B0` and `rootdiskPriv`.

`rootdisk-B0`

> The root file system beginning on cylinder zero will include the critical VTOC block at offset 0 of the disk. UFS file systems and boot blocks are engineered to skip over this critical information, so the VTOC data is protected by the nature of the data on the slice. For a volume, this may or may not be the case. Suppose that the encapsulated boot disk is recycled: `rootvol` and any other subdisks, volumes or plexes on the disk are relocated to other disks, then new

volumes are created from the now-vacant space formally occupied by the root file system. There is no inherent protection for block zero, it is now at risk of being over-written.

VxVM resolves this problem by creating a special subdisk which maps block zero on the root disk, '`rootdisk-B0`'. This subdisk is configured in such a way that it may not be moved or deleted without specific user intervention. With `rootdisk-B0` now guarding block zero of the disk, any new volumes created here cannot overlap that space. `rootdisk-B0` serves no other purpose than to guard the VTOC.

`rootdiskPriv`

Because our boot device was full before encapsulation, VxVM took a small amount of space from the swap device to create the private region. This very likely placed the private region in the middle of the disk, which caused the public and private regions to overlap on this device as was discussed earlier in this section.

VxVM must find some way to ensure that as this disk is re-used as the system ages, no volume is created in that part of the public region which corresponds to the private region's location. It does this by creating an unattached subdisk in this space. The '`rootdiskPriv`' subdisk guards the space in which the private region resides.

Private Region Copies

In order to survive a disk failure, each disk group must manage multiple copies of the configuration database, held within the private regions, on its various component disks. VxVM uses a selection algorithm to maintain some number of configuration copies for resilience, but it does not necessarily place an active config on each disk in the group.

For this reason, you may see some disks with a disabled configuration database section, although the space is still reserved should that disk require an active copy of the database, `configdb`. For absolute maximum resilience, you may wish to force VxVM to keep its configuration database on every available disk with a valid private region. However this comes at some performance cost, the more copies of the config database that the VxVM configuration daemon (`vxconfigd`) must update, the longer those updates will take.

Use "`vxdg list <groupname>`" to view the active config databases and their locations.

```
# vxdg list db_data
Group:     db_data
dgid:      925827691.1088.rtfm
import-id: 0.1220
flags:
version:   20
copies:    nconfig=default nlog=default
config:    seqno=0.1394 permlen=1090 free=1068 templen=15
loglen=165
config disk c1t0d0s2 copy 1 len=1090 state=clean online
config disk c1t3d0s2 copy 1 len=1090 disabled
config disk c1t4d0s2 copy 1 len=1090 state=clean online
config disk c1t5d0s2 copy 1 len=1090 disabled
log disk c1t0d0s2 copy 1 len=165
log disk c1t3d0s2 copy 1 len=165 disabled
log disk c1t4d0s2 copy 1 len=165
log disk c1t5d0s2 copy 1 len=165 disabled
```

Notice that of the 4 disks in this disk group, two have clean and online configuration copies. The other disks have had their configuration database sections disabled. If we would like to enhance the recoverability of this disk group, we may want to force VxVM to write an active config db to each and every disk in the group.

```
# vxedit -g db_data set nconfig=all db
# vxedit -g db_data set nlog=all db
# vxdg flush db_data
# vxdg list db_data
Group:     d_data
dgid:      925827691.1088.rtfm
import-id: 0.1220
flags:
version:   20
copies:    nconfig=all nlog=all
config:    seqno=0.1397 permlen=1090 free=1068 templen=15
loglen=165
config disk c1t0d0s2 copy 1 len=1090 state=clean online
config disk c1t3d0s2 copy 1 len=1090 state=clean online
config disk c1t4d0s2 copy 1 len=1090 state=clean online
config disk c1t5d0s2 copy 1 len=1090 state=clean online
log disk c1t0d0s2 copy 1 len=165
log disk c1t3d0s2 copy 1 len=165
log disk c1t4d0s2 copy 1 len=165
log disk c1t5d0s2 copy 1 len=165
```

Once all configuration regions are declared clean and online, the disk group is able to sustain an arbitrary number of hardware failures without losing the configuration. The data in the public regions may be lost, but if only one disk survives, one may still reconstruct the volume configuration quite easily and then restore the data from backups.

Private Region Size

The private region is the space which VxVM uses to manage itself. It records a number of things within this space: disk header information, configuration data, and an intent log for managing that config database. Due to its limited size, there is only so much configuration data that it can hold. Unfortunately, the size is set at disk initialization time. When the disk is placed under VxVM control, the size of the configuration database which it may hold is fixed for the life of the disk (unless it is re-initialized, see below).

The configuration database is where the volume, plex, and subdisk definitions are stored. This is how VxVM knows how to get to the data in your volumes. When a configuration record changes (such as the creation of a volume or the addition of a mirror to an existing volume), the database must be updated. An intent log (similar to a redo log in a relational database engine) is necessary to keep that database coherent as updates are made. This is the Log section of the private region.

Configuration data is stored as a series of fixed length records in the database. Each object (subdisk, plex, volume, etc) has a 256-byte record in the database which describes its properties. All records occupy the same space on disk, even though object types have property sheets of different sizes. A volume record, for example, has fewer properties than a subdisk record (`sizeof(vol)` < `sizeof(subdisk)`). Even so, it takes one 256-byte slot in the configuration database in order to simplify the data storage and retrieval.

Once a disk group has added objects to fill the space available in the config database (i.e. all slots occupied), no more objects can be created in that disk group. Whether or not sufficient disk space is available to create the volume, if there are no more configuration database records to hold the configuration, the volume cannot be created. This is similar to the concept of slices versus cylinders on a SCSI disk. When all eight slices have been assigned, no more may be created regardless of any unused space on the disk.

The default size of a private region on an initialized, sliced disk is 1024 sectors, rounded up to nearest cylinder boundary. The recent increase in spindle size on disk drives means that a typical cylinder is 3 or 4 times the default requirement of 1024 sectors. There are very few cases where a change in the default private region size is necessary.

A disk may be initialized with a larger than normal private region by invoking `vxdisksetup` with appropriate options. `vxdisksetup` is the low level setup utility invoked by `vxdiskadm` and the vmsa GUI when a disk is placed under VxVM control. While it is possible to enlarge the private region at initialization time, it is frequently inadvisable to do so, a disk set up in this way can not be serviced using `vxdiskadm`.

Consider the scenario of a failed disk, the standard service procedure for replacing this disk will use `vxdiskadm/option4` ("Remove a disk for replacement") followed by `vxdiskadm/option5` ("Replace a failed or removed disk"). To do so in this case will cause the replacement disk to be initialized with the SYSTEM DEFAULT private region size. It will not be initialized with the private region size of the disk it is to replace. This can have lasting effects on the ability of the system to survive many kinds of disk outages. These effects are not reported when the disk is replaced and go largely unnoticed until a problem results.

It is possible to change the system wide default private region length. However, the procedure for doing so is not documented, nor is it supported by the software vendor. Changing the system default in this way removes some of the service problems when disks are replaced, but not all. It merely buries the service problems deeper in the system, making them much more difficult to resolve. Remember that it is rarely necessary to enlarge the private region. The best solution for disk groups with large numbers of VxVM objects is to break the configuration into multiple disk groups. This separates configurations into several smaller databases, each of which will more readily fit into the 1024 sectors VxVM permits.

In cases where you wish to estimate how many objects may be created in a disk group (for planning purposes, for example), it is fairly easy to do so using standard VxVM utilities. You can determine in advance if a disk group will hold a given configuration, so you will know whether to start out with one, two, three, or even more disk groups. A disk group's configuration database is reported by "`vxdg list <groupname>`".

```
# vxdg list TestGroup
Group:     TestGroup
dgid:      943979360.1265.rtfm
import-id: 0.1264
config:    seqno=0.1029 permlen=1092 free=1089 templen=2
loglen=165
config disk c1t3d3s2 copy 1 len=1092 state=clean online
config disk c1t3d4s2 copy 1 len=1413 state=clean online
log disk c1t3d3s2 copy 1 len=165
log disk c1t3d4s2 copy 1 len=165
```

There are several pieces of important information in this output. We will focus on those highlighted with *underlined* typeface.

First, notice that there are two active configuration databases in this disk group. Each has a different length, yet the permanent length of the configuration database for the entire disk group is limited to the smallest of these lengths. This will always be true. Each and every disk in the group must be capable of holding the entire database, so the group limits its size to that of the smallest disk. It is also important to note that the configuration database size for the group is not the sum of all disks. Each disk holds a copy of the database, not a subset of it.

The permanent length of the disk group's configuration database is 1092 sectors. Each sector is 512 bytes. Every VxVM configuration record is 256 bytes in length, implying that each sector will hold exactly two `configdb` records. The sample disk group above is capable of holding a total of 2184 VxVM records (1092 x 2). It is important to distinguish "records" from "volumes" or "disks." Each disk media object, subdisk, plex, and volume (including temporary plexes or subdisks necessary for service procedures) is one VxVM record. This disk group will not hold 2184 volumes.

The above output also suggests that there are 1089 free sectors in its configuration database. There is sufficient room for this configuration database to hold 2178 more records than it currently does. You can assume that there are already 6 records in use (ignoring rounding errors).

To see if the proposed RAID 0+1 configuration will fit into this configuration database, we must estimate the number of records required to represent it. Suppose you wished to create 50 identical volumes, each of which are a 3-way mirror. Each pane of these mirrored volumes (each plex) is a 6-column stripe, the number of records required would be:

50 Volume records
150 Plex records (3/volume)
900 Subdisk records (6/plex)
1100 records sub-total

It is important to note that the number of plex and subdisk records necessary in a volume is dependant upon both the type of volume being created and the number of components to be incorporated into the volume. It should also be noted that volume manager may create components for you, adding to the total number of records. For example, VxVM 3.x introduced the concept of the "layered-volume" used in the creation of RAID 1+0. In a RAID 1+0 volume, additional volume records will be required for the layered-volumes in addition to the plex and subdisk records.

Finally, add to the records sub-total one record for each disk media object necessary to provide sufficient disk space for these volumes. For the purpose of this article assume that there are 90 disks (each disk then touches 10 plexes). The database necessary to support that configuration is estimated to be 1190 records, which is about 600 sectors. This is well within the space provided in the sample disk group above.

Changing the size of the configuration database

Suppose that despite your best planning efforts, you have run out of configuration database space. It is no longer possible to create volume manager objects in a group no matter how much free space available. How can you change the size of the private region to be able to make more volumes? The traditional answer to that question is "Not Possible"; more accurately, “Not Possible without re-initializing every disk in the group.”

The straightforward approach is to:

1. Back up all data to tape
2. re-initialize all disks with larger private region
3. Re-create volumes according to saved configuration
4. Restore data from tape

This is a very lengthy and unnecessary course of action; and, there is another way. However it is more tedious, but it requires no down time and no backup / recovery of the data. One still has to re-initialize all the disks, but this can be done one disk at a time, and still maintain availability of data, assuming that all data is mirrored in a device independent way.

To proceed with this tactic, remove a disk for replacement (i.e. `vxdiskadm/option-4`), initialize it with the larger `privlen`, then replace it (`vxdiskadm/option-5`) with itself. Unfortunately, `vxdiskadm` is not the correct tool for this, because it initializes using an internal function in such a way that one may not specify a larger `privlen`. It uses `vxdisksetup`, so the system wide default is in force.

However, if you really feel you need to do this and you have exhausted all other solutions or workarounds, here is a quick demonstration of the process using the underlying VxVM commands called by `vxdiskadm`. This should be used with extreme caution and only as a last resort:

```
# vxdg list TestGroup | grep ^config
config:    seqno=0.1029 permlen=1092 free=1089 templen=2
loglen=165
config disk c1t3d3s2 copy 1 len=1092 state=clean online
config disk c1t3d4s2 copy 1 len=1092 state=clean online
```

In order to increase "permlen" we have to increase "len" on each config disk. Let's increase on one disk and see what happens:

```
# vxdg -g TestGroup -k rmdisk disk01
# /etc/vx/bin/vxdisksetup -i c1t3d3 privlen=10240
# vxdg -g TestGroup -k adddisk disk01=c1t3d3
# vxrecover -sb -g TestGroup disk01
# vxdg list TestGroup | grep ^config
config:    seqno=0.1031 permlen=1092 free=1089 templen=2
loglen=165
config disk c1t3d3s2 copy 1 len=7827 state=clean online
config disk c1t3d4s2 copy 1 len=1092 state=clean online
```

Notice that the config length on that specific disk jumped way up to 7827 sectors. The permlen didn't change because it must be no larger than the smallest of all active config copies. Disk c1t3d4 is still initialized with the smaller private region. Once disk01 has had its data returned via mirror re-synchronization, we can perform the same remove-reinitialize-replace routine on the other disk in this disk group:

```
# vxdg -g TestGroup -k rmdisk disk02
# /etc/vx/bin/vxdisksetup -i c1t3d4 privlen=10240
# vxdg -g TestGroup -k adddisk disk02=c1t3d4
# vxrecover -sb -g TestGroup disk02
# vxdg list TestGroup | grep ^config
config:    seqno=0.1033 permlen=7827 free=7824 templen=2
loglen=1186
config disk c1t3d3s2 copy 1 len=7827 state=clean online
config disk c1t3d4s2 copy 1 len=7827 state=clean online
```

Notice that once all disks in the group have had their len attribute raised to 7828, the permlen can now be that size. This happens automatically.

There are a number of caveats to using this method:

1. All data has to be mirrored across spindles.
2. Since the private region is increased, the public region must shrink by the same amount. If a disk is 100% used before this process, it won't be able to hold all of the data in its new public region.
3. Subsequent remove/replace operations (such as when a disk fails) using vxdiskadm may not re-initialize with the larger private regions. It will use the system wide default size hard coded into vxdisksetup.

Once again, it should be stressed that this exercise is useful only to illustrate the relationship between `privlen` on each disk and that for the group. It is almost never necessary to create a disk group in which the private region is larger than the default. Because of the service issues involved, it is far more advisable to break the configuration into multiple disk groups rather than increasing the private region length.

Private Region Backup & Recovery

Emergency recoveries involving data are fairly easy to achieve with a backup utility (`ufsdump`, `tar`, `cpio`, NetBackup, Networker, ADSM, etc). With VxVM on the other hand, there is no direct backup facility. One must use `vxprint` and similar reporting tools to fetch configuration data from VxVM, then back up the resulting report files.

Saving VxVM configuration is best accomplished with a script, to ensure consistent and orderly capture of the information. A very brief example script to do this is included here for reference. It stores the resulting backup files locally on the host where the configuration is taken. You will likely wish to copy this information to another host so it will be useful in a full recovery.

```
#!/bin/sh
GROUPS=`vxdg -q list | nawk '$2 ~ /enabled/ {print $1}'`
for G in ${GROUPS}
do
  if [ ! -d /etc/vx/mkfiles/${G} ] ; then
    mkdir -p /etc/vx/mkfiles/${G}
  fi
  vxprint -nv -g ${G} | while read VOLNAME
  do
    echo "## Makefile data saved `date` on host ‘uname -n‘" > \
        /etc/vx/mkfiles/${G}/${VOLNAME}.mk
    vxprint -g ${G} -mhvpsd ${VOLNAME} >> \
        /etc/vx/mkfiles/${G}/${VOLNAME}.mk
  done
done
```

This script collects a ‘`Makefile`’ for each volume in each group imported on the host at the time it is run. These `Makefiles` can be passed directly to `vxmake` to very easily generate an identical configuration with minimal input from the user. VxVM `Makefiles` have the additional advantage that they are ASCII text files formatted in a way that is easy to read and understand. Edits to change key elements of the configuration are quite easy using a standard text editor.

In the cases where you have no saved configuration information, yet need to rebuild according to the VxVM configuration stored in a private region,. it is possible to determine the detailed VxVM configuration by examining the private region directly. There are a number of ways to do this, the most useful of which is to use `vxprivutil`. This is a standard diagnostic utility included with your VxVM package. You will find it installed in `/etc/vx/diag.d`. This utility allows one to report on and manipulate configuration data stored in the private regions of a disk. It is extraordinarily useful in recovery situations. Unfortunately, there is no man page.

The two most useful subcommands for `vxprivutil` are '`scan`' and '`dumpconfig.`' `scan` will report data similar to that of '`vxdisk list`' executed against an imported disk. `vxprivutil` will scan any private region and report what it finds. The `vxdisk` command will report data from the `vxconfigd` database, this only includes information on disks which have successfully been imported as part of their disk group. This makes `vxprivutil`, much more useful for diagnosing or fixing a disk group that won't import.

`dumpconfig` will print to standard output an ASCII version of the raw configuration stored in the private region. While the format is ASCII, it is not very readable; fortunately, `vxprint` understands this as one of the valid database formats. We can pass the data through `vxprint` to generate `Makefiles` in much the same way that we did with `vxprint` in the backup script above.

```
# vxdg import db_data
vxvm:vxdg: ERROR: Disk group db_data: import failed: Disk for
disk group not found
# /etc/vx/diag.d/vxprivutil scan /dev/rdsk/c1t3d3s3
diskid:  943979447.1267.rtfm
group:   name=db_data id=943979360.1265.rtfm
flags:   private noautoimport
hostid:
version: 2.1
iosize:  512
public:  slice=4 offset=0 len=4143520
private: slice=3 offset=1 len=10639
update:  time: 943993750  seqno: 0.14
headers: 0 248
configs: count=1 len=7827
logs:    count=1 len=1186
# /etc/vx/diag.d/vxprivutil dumpconfig /dev/rdsk/c1t3d3s3 | \
             vxprint -D - -mvps > /tmp/vx.make
```

See the `vxprint(1m)` man page for details of the "`-D`" switch and how it is used in this context. This combination of "`vxprivutil dumpconfig`" and the "`vxprint -D -`" filter allows us to generate `Makefiles` directly from data discovered on the private region of any disk under VxVM.

Actually, that is not entirely true. The private region must have been online and active within its disk group; however, not all disks are. This is why we would like all disks in a group to hold an active configuration copy. If all disks are required to hold a configuration copy, any one of them could be used to reconstruct the entire disk group's volume configuration. It need only be scanned and have its configuration written to a `Makefile`.

Conclusion

The VxVM private region is key to the proper and efficient functioning of VxVM. This article has covered: the function of the VxVM public and private regions, the location of the private region and it's role in providing resilience, and the special considerations for root disk encapsulation. Additionally, the mechanism that VxVM uses for configuration database sizing and why the defaults should never be changed were examined.

By understanding the private region, the System Administrator will gain insight into how VxVM functions and why it makes the decisions it does. Plus, a greater understanding of VxVM will provide the System Administrator with the ability to configure their storage for maximum reliability while minimizing the amount of time the System Administrator needs to effect recovery.

Author's Bio: Gene Trantham

Gene Trantham is a Staff Engineer for Enterprise Engineering at Sun Microsystems. Prior to joining Sun, he spent eight years as a Unix® system administrator specializing in storage management and disaster recovery. While at Sun, Gene has spent most of his time in the field, servicing storage and high-end servers at some of Sun's largest accounts.