Another Kind of Volume Manager

CHAPTER 5: VOLUMES

by Volker Herminghaus

5.1 Overview

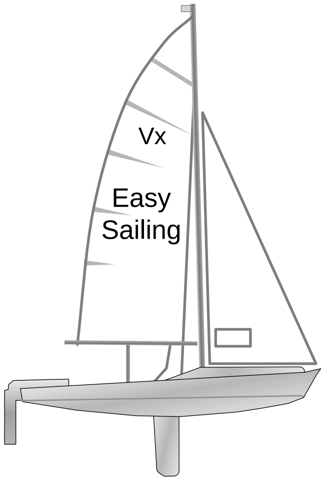
In the previous chapter you learned a lot about disk groups, or DGs. But DGs are mere containers for the truly interesting virtual objects, namely volumes. In this chapter we will concentrate on volumes: how they are created, modified, and removed. How you put a file system on them and use them for productive work. How they behave on disk, path or host failures. How I/O to a mirrored volume is handled and how consistency is always guaranteed. What internal virtual objects a volume consists of and how you can inspect and understand them, and much more.

5.1.1 What is a Volume?

A VxVM volume is a modern replacement for a slice or partition. It serves as a container for file system or raw data and allows I/O operations to proceed without being volume-aware, i.e. the software layers above the volume need not be changed in order to work with volumes. Volume I/O is completely transparent unless extra effort is undergone to find out the exact nature of the device. So for all purposes a volume can replace a Solaris or Linux partition, an AIX volume or an HP-UX volume (especially since HP's internal volume management is based on an old version of Veritas Volume Manager). Be aware that a VxVM volume is **not** the virtualized equivalent of a **hard disk**, however! If it was, then it could have a VTOC in block zero and you could partition it or use **vxdisksetup** or **vxencap** on it. That wouldn't make too much sense, would it? Instead of emulating a hard disk it emulates a partition. And what is a partition? A partition is a block-addressable extent that stores data persistently, nothing else. So it should be something relatively easy to emulate.

But there is one crucial feature of a partition that is not so easy to emulate in a volume management software, and that is the following: A partition, since it holds just one copy of the data, will always deliver the same data for any given extent unless you update the data. In a redundant volume, let's say a three-way mirror, this is not at all easy to emulate. Of course you write data to all three mirrors when a write is issued to the volume. But will they all succeed? What if there are hundreds of I/Os outstanding on each of the mirrors and then the host crashes. After a reboot, how can we be sure that all mirror sides are identical? Is there a preferred mirror side that gets written first and holds the most recent data? No, there isn't! Does VxVM copy one mirror side over all the others? No, it doesn't! Does VxVM compare the mirror sides? No, it doesn't! Does VxVM break the partition paradigm and actually deliver different data on consecutive reads of the same extents? No, it doesn't. Well, how does VxVM the guarantee conformance to the critical partition paradigm: to give exactly the same data for the same extent unless the extent is updated?

The answer to this question has to do with quantum physics and Schrödinger's cat (really!). This answer, and much more, will be discussed in detail in the sections that follow. But first, some simple stuff.



5.2 Simple Volume Operations

5.2.1 Creating, Using and Displaying a Volume

All common volume operations are done via one simple, unified command. This command is called **vxassist**. In very early versions of VxVM volumes were created by allocating extents from disks in a disk group, associating them with plexes, and putting them into volume objects. But that was much too complicated and cumbersome to be useful. So the command **vxassist** was created, which did all the magic of allocating virtual objects, calculating disk and plex offsets, mapping everything together and starting the volume. In fact, you can still watch part of what **vxassist** does internally when you pass the **-v** switch to the command. You can then see what other tools are called by **vxassist** internally. Although the allocation commands are usually not displayed because allocations are done by calls to the vxvm allocation library. Let us start by creating a DG out of six disks. Let's call the DG "**adg**" for "a disk group". Then, let's create a simple volume "**avol**" (for "a volume") in our DG using the **vxassist** command. We'll put a file system on the volume and mount the file system. Keep in mind what you learned in chapter 2 (Exploring VxVM) that the device driver for VxVM resides in **/dev/vx/*dsk/<DGname>/<Volname>**:

```
# vxdisk list # check what we have: six VxVM disks in cdsdisk format
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t3d0s2     auto:cdsdisk    -            -            online
c0t4d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online
c0t12d0s2    auto:cdsdisk    -            -            online
# vxdg init adg adg01=c0t2d0 adg02=c0t3d0 adg03=c0t4d0 (...) # make a DG
# export VXVM_DEFAULTDG=adg # Set our default DG for this session
# vxassist make simplevol 1g # Create a volume of exactly 1 GB
# newfs /dev/vx/rdsk/adg/simplevol # Make a UFS file system on the volume
newfs: construct a new file system /dev/vx/rdsk/adg/simplevol: (y/n)? y
/dev/vx/rdsk/adg/simplevol:    2097152 sectors in 1024 cylinders of 32 tracks,
64 sectors
        1024.0MB in 32 cyl groups (32 c/g, 32.00MB/g, 15872 i/g)
super-block backups (for fsck -F ufs -o b=#) at:
 32, 65632, 131232, 196832, 262432, 328032, 393632, 459232, 524832, 590432,
 1443232, 1508832, 1574432, 1640032, 1705632, 1771232, 1836832, 1902432,
 1968032, 2033632,
# mount /dev/vx/dsk/adg/simplevol /mnt # Mount the file system (FS)
# df -h /mnt # Check the FS: 1MB used, 903MB free
Filesystem   size used  avail capacity  Mounted on /dev/vx/dsk/adg/simplevol
            961M  1.0M  903M    1%    /mnt

# cp -r /kernel/* /mnt/. # Put some data on the volume
# df -h /mnt # Check the FS: 89MB used, 815MB free
Filesystem   size used  avail capacity  Mounted on /dev/vx/dsk/adg/simplevol
            961M  89M  815M    10%    /mnt
# ls /mnt
crypto     drv        exec       ipp        lost+found  misc        strmod
dacf       dtrace     fs         kmdb       mac        sched        sys
```

OK, the volume seems to work just like a normal partition. Now let's look at the volume. First, check what the volume looks like in the file system tree, i.e. what attributes the device file has:

```
# ls -l /dev/vx/*dsk/adg/simplevol
brw-------   1 root root  270, 62000 May 16 21:15 /dev/vx/dsk/adg/simplevol
crw-------   1 root root  270, 62000 May 16 21:16 /dev/vx/rdsk/adg/simplevol
```

OK, we see a **block** and a **character** device. Compare these to normal disk partitions:

```
# ls -lL /dev/*dsk/c0t10d0s2
brw-r-----   1 root     sys       32, 74 May 17 01:48 /dev/dsk/c0t10d0s2
crw-r-----   1 root     sys       32, 74 May 17 01:48 /dev/rdsk/c0t10d0s2
```

They look just the same except the volume is not group readable by **sys** and has different major and minor numbers. We actually cannot see a lot about the volume in the file system representation of a volume. We cannot see what virtual objects it consists of, or what state they are in. Why can't we see that in the file system? Because, as was stressed before, the volume has to look exactly the same as a partition to the rest of the system, so there is simply no file system interface to look into the details of the volume. Can we derive much about the internals of a **partition** by looking at the disk **partition** device node? No, we cannot. With volumes it is just the same. The device node is not the right place to look in order to find more information about the storage object.

What we need to do is ask VxVM to show us what it knows about the volume. There is a command for that purpose: it is called **vxprint** and normally takes the DG as a parameter (it does **not** respect the default-DG). It shows all virtual objects residing in the DG, or all virtual objects from all DGs if no DG is specified.

```
# vxprint -g adg
TY NAME         ASSOC        KSTATE   LENGTH   PLOFFS  STATE    TUTIL0  PUTIL0
dg adg          adg             -        -        -       -        -       -

dm adg01        c0t2d0s2     -        17846208 -        -        -       -
dm adg02        c0t3d0s2     -        17702192 -        -        -       -
dm adg03        c0t4d0s2     -        17616288 -        -        -       -
dm adg04        c0t10d0s2    -        17616288 -        -        -       -
dm adg05        c0t11d0s2    -        8315008  -        -        -       -
dm adg06        c0t12d0s2    -        35771008 -        -        -       -

v  simplevol    fsgen        ENABLED  2097152  -        ACTIVE   -       -
pl simplevol-01 simplevol    ENABLED  2097152  -        ACTIVE   -       -
sd adg01-01     simplevol-01 ENABLED  2097152  0        -        -       -
```

The output of the vxprint command is easy to parse if you know what to look for, so let's go through it in this Easy Sailing part. If you are bored, you can skip to The Full Battleship part beginning on page 120.

5.2.2 Useful vxprint Flags Explained

In the later parts of this book we will use several more flags to make the output more palatable to the reader. These flags are:

- **-t** to display each line in its individual format, with a header at the beginning of each disk group record explaining what the meaning of each column of output is for the given object
- **-q** to suppress the header information (once you get used to the output format you don't need the headers any more, and they take away a whole lot of space)
- **-h** to hierarchically list all objects below the specified ones (i.e. if you specify volumes it will also list the plexes and subdisks contained inside the volumes)
- **-v** to display the volumes and, due to the -h flag above, all records associated with them. If we did not specify -v, or if we used -r (related) instead of -h, then the output would also list the disk group and disk media information, which we like to suppress here in order to save valuable page space.

Basically, the output consists of several sections: a header section, a disk group section, a Disk Media section, and a Volume section. The header section in the example above consists of only a single line listing Type, Association, KernelState, Length, PlexOffset, User State, and TemporaryUtility and PermanentUtility fields. The header section will be expanded in the next command to show much more detail.

The DG section is right below the header and starts with the abbreviation for disk group, "**dg**". As you can see, it is associated to itself and has neither state nor other information to display. The Disk Media section is below that and has lines starting with the abbreviation for Disk Media, "**dm**". It shows what access names are associated with our VxVM disk names, and their length. Everything else that appears in the header line is inapplicable, which is indicated by a dash ("–").

The last part is the most interesting: It shows three objects: The volume (**simplevol**, in the line beginning with a "**v**" for volume), a plex called **simplevol-01**, its line beginning with "**pl**" for plex. It is associated with the volume, as you can see from looking at the **ASSOC** column of the plex record. And finally a record beginning with "**sd**" for subdisk and listing the attributes of the subdisk **adg01-01.** The name implies that it is the first subdisk that was allocated from the disk medium **adg01**. This subdisk is associated with the plex **avol-01**. Again, you can check this by looking at the value in its ASSOC column. The **vxprint** command knows the association hierarchy and arranges the objects accordingly: The volume at the top level, then the plex, then the subdisk.

All of these objects have kernel state (**KSTATE**) information (**ENABLED**) etc., but still many of the fields are inapplicable in this output format.

To improve on the output format, we suggest you use the **-r** and **-t** flags for vxprint. This will use an individual format and show individual headers for each object type and thus pack much more useful information into the lines. Later, when you are familiar with the output format, you can add the **-q** flag, which will suppress the large block of headers at the beginning.

Now that we have seen all we wanted to see by using **vxprint** let's get rid of the volume to create some new ones. There are several ways to do remove a volume: one is using **vxassist remove volume <volname>** and one is using the more low-level **vxedit -rf rm <volnames>** command. The **vxedit** command actually has several advantages: it involves less typing, it can remove more than one volume at a time, and is can remove volumes regardless of their status, while **vxassist** tends to have problems removing volumes that are somehow mangled, have associated snapshots, don't have all plexes enabled and such. Here are the two alternative commands, choose for yourself:

```
# vxassist remove volume simplevol
# vxedit -rf rm simplevol
```

You may wonder what the **-rf** flags do that we pass to **vxedit.** The short explanation is that **vxedit** will simple remove any virtual object you pass it. But if that object has associated subobjects, then **vxedit** will need to recurse into the subtree and remove all the associated objects recursively, from bottom to top. That is the reason for supplying the **-r** flag, because that will enable recursive deletion. If the volume is currently started (i.e. it is **ENABLED** and **ACTIVE**) then **vxedit** also requires the use of the **-f** flag (force). You can omit the **-f** flag if the volume is stopped.

If you tried the commands you will have noticed you get an error message:

```
VxVM vxedit ERROR V-5-1-1242 Volume simplevol is opened, cannot remove
```

This is because the volume is still mounted. Once you unmount it you can delete the volume. Rest assured that VxVM keeps you from deleting objects that are currently in active use, just like the **format** command prohibits you from deleting a currently mounted partition. by telling you: "**Cannot label disk while it has mounted partitions**".

So we unmount the volume and delete it:

```
# umount /mnt
# vxedit -rf rm simplevol
```

This time it works! Now you know how to create, use, inspect, and remove volumes from a disk group. Time for some more interesting topics.

5.2.3 Starting and Stopping Volumes

VxVM's state machine

When a volume becomes available to the system (via creation or via DG import) the system does not know if the contents of the volume can or cannot be trusted. For instance, a volume on a DG that was deported and is now imported might have had write errors while it was imported by another host. Or the other host could have crashed while it was writing to a redundant volume etc. It is obvious that some kind of check needs to be done before we can trust the volume's integrity.

This is analogous to what the **mount()** system call needs to do when a user wants to mount a file system: can the contents of the file system be trusted? Was the file system properly unmounted, thus flushing all buffers and leaving it in a proper state? Or had the system crashed and left the file system in an inconsistent state? In classical UNIX, the **mount()** and **umount()** system calls use a single bit, the **dirty flag**, which is persisted in the file system's superblock, to remember the state of the file system. When a file system is mounted, the dirty flag is inspected and, if it is clean, the file system is mounted and the dirty flag set to **dirty**. When the file system is unmounted, the dirty flag is reset to **clean**. If a system crashes, then the dirty flag is, of course, not reset and remains dirty. So the next time a user tries to mount the file system, the **mount()** system call inspects the dirty flag and finds it **dirty**. In that case, the file system is not mounted by a file system check (**fsck**) is requested.

The same kind of check must be done by volume manager, albeit on a lower level, to ensure that a mirrored volume is internally consistent. This check does by no means imply reading and comparing all mirrors, just like mounting a file system does not always require a file system check (**fsck**). That would be very wasteful. Instead, VxVM maintains state information in the configuration database that get changed when certain events happen. For instance, when a volume is written to the first time after opening it, a bit in the configuration database is set that flags the fact that the device is now potentially out of sync. When the volume is closed (which can only be done after all I/O has been flushed to all mirrors) this bit is reset. This can be directly compared to . There is more state information like this. For instance, a plex that has encountered unrecoverable I/O errors to one of its subdisk is marked DISABLED so that no further read (or write) traffic for the plex is generated. Then, when the subdisk is repaired, the plex state is set to STALE, indicating that the data inside the plex is not current.

When you want to use a volume then all this state information needs to be checked to find out if I/O to the volume is possible and reasonable, or if e.g. the mirrors are out of sync. This process needs to be done only once, at the beginning. Whatever happens while the volume is online and undergoing I/O will be caught by VxVM and lead to appropriate reactions. But the initial state must be checked before using the volume. Exactly that is done by the **vxvol start** command. It is a volume sanity check followed by allowing I/O if the volume is sane. It does so by setting the volume's state to ACTIVE and its kernel state to ENABLED.

So what does **vxvol stop** do? Functionally it does nothing but set the ENABLED flag off so that further user access is impossible. It is neither necessary nor required by VxVM to stop your volumes before deporting a DG. It does make some maintenance commands easier that otherwise require an extra option flag to force an action in order to work on a started volume, but that's about it. You don't normally stop any volumes, nor do you normally need to.

There is much more about the state machine in the troubleshooting sections of this book beginning on page 349.

5.3 Volume Layouts and RAID Levels

In this section you will learn how to specify the various RAID levels for VxVM volumes. The first thing you will learn is that VxVM does not actually use RAID levels numbered 0 through 5. Instead, it uses what would properly be called volume features: The striping feature, the mirroring feature, the XOR-parity feature, the concatenation feature, the logging feature and so on. You will see that these features can be added to make up almost any reasonable volume type. Not all combinations are possible, but then again, not all are reasonable either, and what's possible should make you a pretty happy administrator indeed.

5.3.1 Volume Features Supported by VxVM

Concatenation

This is the simplest layout of all the RAID levels. Originally called RAID-0, it was intended to connect disks together to overcome the size limitation. By concatenating disks, the capacity of the resulting volume was equivalent to the sum of the capacities of the individual disks. Disks could be of different sizes.

Of course, VxVM will not concatenate **disks** to form a larger virtual disk, like the original RAID concept did. Instead, it will concatenate **subdisks** (extents on physical disks) by mapping them into a plex at different plex offsets (which turns up as **PLOFFS** in vxprint), resulting in a contiguous virtual address space that can grow to almost unlimited size. There is no need to specify anything particular on the command line for VxVM to enable concatenation. The concat volume layout is the default layout unless the **/etc/default/vxassist** file forces a different layout. VxVM will concatenate as necessary, provided there is enough space left that has the required storage attributes. We will say a little more on storage attributes later.

Synopsis and sample command to create a volume with the concatenation feature:

```
# vxassist -g <DGname> make <volname> <size> [layout=concat]
# vxassist -g adg make avol 1g
# vxprint -htv -g adg

v  avol         -           ENABLED  ACTIVE   2097152  SELECT    -       fsgen
pl avol-01      avol        ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg01-01     avol-01     adg01    0        2097152  0         c0t2d0  ENA
```

The first line starts with the letter **v**. This means it is a line describing the volume virtual object. The next line starts with the string **pl**, which means that this line describes a plex virtual object. The third line describes a subdisk, which can be identified by the string **sd** at the beginning of the line. The second columns of each line describes the virtual object's name. And the third column of each line of the output identifies what other virtual object the given object is associated to. The volume, as we can see, is not associated to anything; its third column contains a dash (-). The plex, however (whose name is **avol-01**), is associated to an object called **avol**, which happens to be the volume that the plex resides in. And the subdisk (**adg01-01**) is associated with the plex **avol-01.** We emphasized these words to make you find the relevant information more quickly.

Note that the sizes in column six of volume, plex, and subdisk are identical to what we specified, but are given in blocks of half a kilobyte each. So in order to find the amount of megabytes we have to divide by 2 and then by 1024, or just simply by 2048:

```
# bc -l
2097152/2048
1024.0000000000 # Exactly what we wanted: 1024 MB equals 1.0 GB!
```

Other information you will find useful to know is the following:

The word SELECT in column 7 of the volume line means the read policy. This could be SELECT, ROUND, and PREFER. More information on volume read policies can be found later in this chapter. In the same column of the plex line you will see the word CONCAT. This designates the layout of the plex (there is actually no such thing as a volume layout - a plex has all the layout; the volume just has one or more plexes, turning it into a mirrored or unmirrored volume).

In column four and five the virtual objects plex and volume display a kernel state and a (user) state. A state of ENABLED means that I/O to the object is possible, while DISABLED means no I/O is possible to the object at all. There is also a third kernel state, namely DETACHED, which means that user I/O is not possible but kernel I/O is possible. This is needed for internal resynchronisation or RAID-5 initialization mechanisms and not of general interest here.

If we stop the volume, or before the volume is started after importing the disk group, the kernel states will be disabled and I/O will not be possible:

```
# vxvol stop avol # stop the volume
# vxprint -qhtvgadg # look what's changed
v  avol         -            DISABLED CLEAN    2097152  SELECT    -       fsgen
pl avol-01      avol         DISABLED CLEAN    2097152  CONCAT    -       RW
sd adg01-01     avol-01      adg01    0        2097152  0         c0t2d0  ENA

# newfs /dev/vx/dsk/adg/avol # trying to use the volume fails
/dev/vx/rdsk/adg/avol: No such device or address
# vxvol start avol # start the volume again
# vxprint -qhtvgadg # DISABLED has become ENABLED; CLEAN has become ACTIVE
v  avol         -            ENABLED  ACTIVE   2097152  SELECT    -       fsgen
pl avol-01      avol         ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg01-01     avol-01      adg01    0        2097152  0         c0t2d0  ENA
# newfs /dev/vx/dsk/adg/avol # using the volume now works
newfs: construct a new file system /dev/vx/rdsk/adg/avol: (y/n)? y
/dev/vx/rdsk/adg/avol:  2097152 sectors in 1024 cylinders of 32 tracks, 64 sec-
tors
        1024.0MB in 32 cyl groups (32 c/g, 32.00MB/g, 15872 i/g)
super-block backups (for fsck -F ufs -o b=#) at:
 32, 65632, 131232, 196832, 262432, 328032, 393632, 459232, 524832, 590432,
 1443232, 1508832, 1574432, 1640032, 1705632, 1771232, 1836832, 1902432,
 1968032, 2033632,
```

Striping

Striping originated as RAID-0, just like the concat layout. To prevent misunderstandings: the RAID levels were more specifically called **RAID-0 concat** and **RAID-0 stripe**, respectively. The RAID levels have been discussed in chapter 1 on page 7-9 to some extent, but let's reiterate the basics here.

A striped volume distributes its contents in a regular pattern over its subdisks. The intention of striping originally was to alleviate the long wait times that occurred on large read/write I/Os to slow disks. These disks' controllers or the HBAs were so slow that the disks were often formatted with an interleaving factor because data could not be transferred at full speed to the host. Striping is a little complicated to explain in words, but it may be okay if we use a somewhat creative analogy here:

Imagine you had ten salamis, and you want to make salami sandwiches. The salamis are your separate disk spindles and the sandwiches are your volumes. Then if you make a concat volume, you slice up the first salami and put the slices on the sandwiches until the first salami is gone, then you take the second salami and so on. If you were VxVM, by the way, then you would try cutting each salami at exactly the right angle so that a single slice covers the whole sandwich, while IBM's AIX LVM would cut the salamis into lots of tiny slices and then cover the sandwich with them but that's not the issue here.

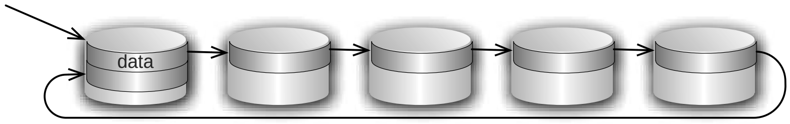


Figure 5-1: Striping is reminiscent of making a salami sub. Data is mapped in identically sized slices, one slice at a time, across multiple "columns". The size of a data slice is called stripe size, stripe unit size, stripe width, or similar. There is no generally accepted term for the size of one "layer of salami", i.e. the size of a slice times the number of columns.

When you are striping then you are taking tiny slices from each salami and putting them on your sandwiches in a row, so that the slice from salami two follows salami one, salami three follows salami two, salami 4 follows salami three, and so on, until your last salami's slice is followed by a slice from salami one again. The number of salamis you are using is the number of **`columns`**, while the size of the individual slices is called the **`stripe width`** or **`stripesize`**. This ensured the taste is about average even if all your salamis are different, and that all salamis are used more or less evenly regardless of which way you bite the sandwich.

The default stripesize for VxVM is 64 KBytes (which BTW is far too small - 1MB would be a lot better but it is probably best not to stripe at all nowadays).

Striping has become much more common recently due to the advent of storage arrays that stripe internally, and of both system administrators and database administrators becoming more familiar with the concepts. However, most of them have forgotten Moore's law and the problem of mechanics, and therefore get most of their decisions wrong when it comes to storage layout.

Striping may be one of the most misused features of all RAID systems, so please make sure you know what you are doing by reading and fully understanding the appropriate discussion on volume formats beginning on page 137. The short version is this: Striping was great 20 years ago. Striping is often a bad idea today when you are using JBOD disks (because Moore's Law has been working against you for 20 years or so). Striping may still improve your individual performance if your data are on a storage array, but it will invariably do so at the cost of reducing the performance for everybody else who is using the same storage array. Read more about that later this chapter where we compare volume formats.

Here is the synopsis and a sample command to create a volume with the striping feature:

```
# vxassist -g <DGname> make <volname> <size> layout=stripe [ncol=<x>] …
# vxassist -g adg make avol 1g layout=stripe ncol=5 stripewidth=2048
# vxprint -qhtvgadg
v  avol        -           ENABLED  ACTIVE   2097152  SELECT    avol-01  fsgen
pl avol-01     avol        ENABLED  ACTIVE   2099200  STRIPE    5/2048   RW
sd adg01-01    avol-01     adg01    0        419840   0/0       c0t2d0   ENA
sd adg02-01     avol-01      adg02    0        419840   1/0        c0t3d0   ENA
sd adg03-01     avol-01      adg03    0        419840   2/0        c0t4d0   ENA
sd adg04-01     avol-01      adg04    0        419840   3/0        c0t10d0  ENA
sd adg05-01     avol-01      adg05    0        419840   4/0        c0t11d0  ENA
```

Note the layout in the **plex** line (line 2) is now **STRIPE**, with the values of **5/2048** for the number of columns and the stripe width. Below the **STRIPE**, you can see which column each subdisk belongs to, e.g. 3/0 means the subdisk belongs to column three and it mapped into the striped plex at offset 0.

Mirroring

Mirroring originated as RAID-level 1 and was the first layout that created redundant volumes. As always, in contrast to the original RAID concepts, VxVM will never mirror a physical disk onto another physical disk. Instead it will allocate subdisks for two plexes and put two plexes into a volume, resulting in the data to be written into each plex and thus being redundant. Mirroring can be done by specifying the "**mirror**" attribute to the layout subcommand in **vxassist**. It can also be added after the fact by issuing **vxassist mirror <volname>** with an already created volume. Mirroring can be done with up to 32 mirrors and mirrors can be added and removed transparently while the volume is online and undergoing I/O. The VxVM allocation library will automatically arrange storage allocation such that no two mirrors ever share the same disk, as that would reduce both performance and redundancy of the volume. If you want more than simple redundancy (that would be equivalent to a number of two mirrors) you can specify a different number of mirrors using the **nmirror** subcommand.

Synopses and sample commands to create a volume with the mirroring feature:

```
# vxassist -g <DGname> make <volname> <size> layout=mirror [nmirror=<x>]
# vxassist -g adg make avol 1g layout=mirror [nmirror=3]

# vxassist -g <DGname> make <volname> <size> nmirror=<x>
# vxassist -g adg make avol 1g nmirror=3
vxprint -qhtvgadg
v  avol         -            ENABLED  ACTIVE   2097152  SELECT     -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   2097152  CONCAT     -        RW
sd adg01-01     avol-01      adg01    0        2097152  0          c0t2d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT     -        RW
sd adg02-01     avol-02      adg02    0        2097152  0          c0t3d0   ENA
pl avol-03      avol         ENABLED  ACTIVE   2097152  CONCAT     -        RW
sd adg03-01     avol-03      adg03    0        2097152  0          c0t4d0   ENA
```

In this output all plex lines were emphasized in order for you to understand the volume structure more easily. Since each data plex is a container for the whole volume's contents, with three plexes you have three containers and accordingly a three-way mirror. Not that the layout of each plex is **CONCAT**, because inside the plexes there is neither striping nor RAID-5.

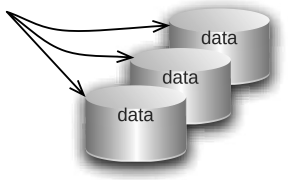


Figure 5-2: Mirroring means more than instance of the data is stored in the volume. One instance of the data is stored in each data plex. Each data plex is either striped or concatenated (RAID-5 plexes in a mirror are not supported, although that could be forced with a little trickery). In this picture, each row constitutes one concat plex in a three-way mirror. This is what the above `vxprint` sample output displays: three plexes, each plex has a `CONCAT` layout, each plex has a single subdisk mapped inside it.

RAID-5

RAID-5 is actually nothing but a stripe, but with a special extra column that holds a lossless checksum of all data columns. The extra columns is called the parity column, because each bit in this columns holds information about the parity of the sum of the corresponding bits in all data column. If any of the data columns fails, then VxVM can use the parity columns to regenerate the original data on the failed column by combining all data plus parity using an XOR operation. The details of this are discussed in The Full Battleship. Suffice it to say that a parity-protected stripe can lose one column and still work, no matter how many columns there are. Loss of a second disk, however, will result in loss of volume integrity and therefore loss of data. The predecessor to RAID-5, RAID-4, used a dedicated column (or disk) for parity, which led to a bottleneck on multi-threaded writes. RAID-5 overcomes this bottleneck by distributing the parity information across all columns on a stripesize-by-stripesize basis.

RAID-5 is a feature that you will not want to use any more once you know how it needs to be done in a host-based system. No matter which way you look at it, RAID-5 will be slow on small writes. It will be unreliable after a single disk failure. And it will be slowed down after a disk failure and made more unreliable because of that failure in addition to being extra slow. It also needs one disk more than you think it needs, because running RAID-5 in software without a transaction log is irresponsible, so it loses part of the price advantage it has against mirroring. In short, we can almost guarantee that after you read the part of volume layouts you will never even think about deploying RAID-5 in an enterprise. It can still make a lot of sense in SOHO or university environments where cost needs to be as low as possible and uptime is not the most critical factor. But you will not deploy it in a financial institution if you like your job there.

The number of columns that is given on the command line does include the parity column but not the additional log plex that is highly recommended for RAID-5 and that is automatically added unless you prohibit it. So you need at least one more disk than the columns specification (one per column plus one for the log, which must not reside on any of the columns.

Synopsis and sample command to create a volume with the concatenation feature:

```
# vxassist -g <DGname> make <volname> <size> layout=raid[5] [ncol=<x>] …
# vxassist -g adg make avol 1g layout=raid5 ncol=5 stripesize=256k
# vxprint -qhtvgadg
v  avol        -           ENABLED  ACTIVE   2097152  RAID      -        raid5
pl avol-01     avol        ENABLED  ACTIVE   2097152  RAID      5/512    RW
sd adg01-01    avol-01     adg01    0        524288   0/0       c0t2d0   ENA
sd adg02-01    avol-01     adg02    0        524288   1/0       c0t3d0   ENA
sd adg03-01    avol-01     adg03    0        524288   2/0       c0t4d0   ENA
sd adg04-01    avol-01     adg04    0        524288   3/0       c0t10d0  ENA
sd adg05-01    avol-01     adg05    0        524288   4/0       c0t11d0  ENA
pl avol-02     avol        ENABLED  LOG      76800    CONCAT    -        RW
sd adg06-01    avol-02     adg06    0        76800    0         c0t12d0  ENA
```

You will notice that there are two plexes here. But this does not mean that the volume is mirrored. Only one of the plexes is marked **ACTIVE**, while the other one is marked **LOG**. This means that the plex does not contain user data, but log data, which is just a very small amount compared to the volume size. Only data plexes count for redundancy, not log plexes!

But you will notice a few more things: The volume's read policy (column seven of the first line) is neither SELECT, ROUND, or PREFER, but a new policy: RAID. And this is the same as the data plex's layout. Apart from that, the data plex looks just like a standard five-column stripe.

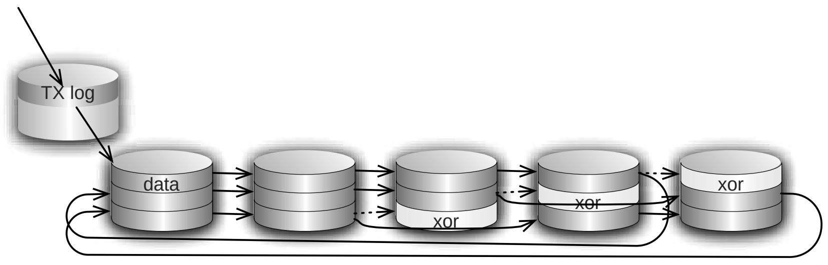


Figure 5-3: This is what the above **vxprint** sample output displays. The plex marked **RAID** in the **vxprint** output consists of five columns, four of which are actually available as net capacity for user data. One fifth is lost to the distributed parity information which is spread across all columns in left-symmetric layout (simplified to just three clusters marked "XOR"). The second plex, which is marked **LOG** in the **vxprint** output is actually a small concat plex. But it does not hold a copy of the volume data. Instead, the RAID-5 write method uses it to store the five most recent write transactions to the volume. This makes recovery faster and the volume more reliable in special cases..

Combining Volume Features

Using VxVM it is really easy to combine volume features. For the basic layout parameters, just pass them in a comma-separated list on the command line. Other variables, like the number of mirrors, logs, or columns, can be passed as key/value pairs similar to what we've already been doing when naming disk media (e.g. **adg01=c0t2d0**). We will discuss a large number of such parameters in The Full Battleship of this chapter.

Synopsis and sample command to create a volume with combined features:

```
# vxassist -g <DGname> make <volname> <size> [layout=<feature0>,<feature1>,…]
# vxassist -g adg make avol 100m layout=stripe,mirror,log nlog=2 nmirror=3
# # vxprint -qhtvgadg
v  avol         -           ENABLED  ACTIVE   204800   SELECT    -        fsgen
pl avol-01      avol        ENABLED  ACTIVE   204800   STRIPE    2/128    RW
sd adg01-01     avol-01     adg01    0        102400   0/0       c0t2d0   ENA
sd adg02-01     avol-01     adg02    0        102400   1/0       c0t3d0   ENA
pl avol-02      avol        ENABLED  ACTIVE   204800   STRIPE    2/128    RW
sd adg03-01     avol-02     adg03    528      102400   0/0       c0t4d0   ENA
sd adg04-01     avol-02     adg04    0        102400   1/0       c0t10d0  ENA
pl avol-03      avol        ENABLED  ACTIVE   204800   STRIPE    2/128    RW
sd adg05-01     avol-03     adg05    528      102400   0/0       c0t11d0  ENA
sd adg06-01     avol-03     adg06    0        102400   1/0       c0t12d0  ENA
```

```
pl avol-04      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg05-02     avol-04      adg05    0        528      LOG       c0t11d0  ENA
pl avol-05      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg03-02     avol-05      adg03    0        528      LOG       c0t4d0   ENA
```

This is the last example of the simple volumes that we will discuss in detail. The plexes are again emphasized. Note that each plex shows up as a two-column stripe with 128 blocks stripewidth, and there are three data plexes total (the first three). The other two plexes are of type **CONCAT**, are very small (528 blocks), and are designated as **LOGONLY** plexes, which means they hold a dirty region log. We will deal with an in-depth discussion of logs in a later chapter.

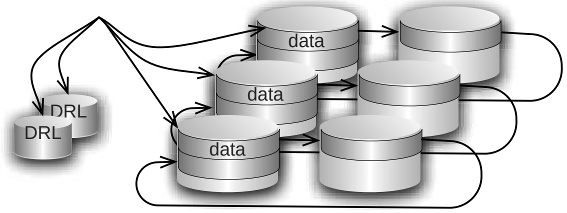


Figure 5-4: The volume from the above `vxprint` sample output looks like this picture. It consists of three plexes, each marked as `STRIPE` in the `vxprint` output. Each of these corresponds to one of the rows in the right hand part of this picture The last two plexes, marked `LOGONLY` in the `vxprint` output, constitute a tiny, mirrored bitmap called the Dirty Region Log, or DRL. The DRL is used to improve startup behavior after system crashes and is discussed in the log chapter (chapter 7).

5.4 Volume Maintenance

Adding a Mirror

Adding a mirror is easy with VxVM: just tell **vxassist** to mirror the volume. This will create another plex and attach it to the volume. The plex attach action will initialize the new plex with the volume's contents. If you specify a number of mirrors/plexes to add, that number of plexes will be added to the volume. It does not matter whether the volume was already mirrored or not, or if it is a concatenated or striped volume. RAID-5 volumes can not be mirrored this way, at least not in a single **vxassist** command.

Synopsis and sample command to add a mirror to a volume

```
# vxassist -g <DGname> mirror <volname> [nmirror=<x>]
# vxassist -g adg mirror avol nmirror=3 # will add three mirrors
```

Removing a Mirror

Removing a mirror is almost as simple as adding one. It just requires passing an extra command word to **vxassist**, namely **remove mirror** instead of **mirror**. You can also specify a number if mirrors to the remove command, but this will not work as expected: while when adding mirrors the **nmirror=<x>** parameter specifies the number of new mirrors to add, in the case of removing it specifies how many mirrors are to remain!

Synopsis and sample command to remove a mirror from a volume

```
# vxassist -g <DGname> remove mirror <volname> [nmirror=<x>]
# vxassist -g adg remove mirror avol nmirror=3 # will leave three mirrors
```

Adding a Log

Just like mirrors, you can add a log to a volume by simply passing the right parameter to vxassist. The keywords here are **addlog**, **nlog=<x>,** and **logtype={dco|drl}.** The nlog parameter defines the number of logs to add to the volume, and the **logtype** parameter defines whether it is going to be a DRL (dirty region log) or DCO (data change object)

We have not discussed what each type of log is or does. There are various kinds of logs., and they are discussed in chapter 7 beginning on page 173. Because this book is also recommended for looking up frequent procedures we chose to put the part about managing logs here where it belongs, and explain it later to those readers who prefer to read the book as a technical training guide.

Synopsis and sample command to add a log to a volume

```
# vxassist -g <DGname> addlog <volname> [nlog=<x>] [logtype={dco|drl}
# vxassist -g adg addlog avol nlog=2 # will add two log plexes
```

Removing a Log

Like a mirror, a log can be removed by specifying **remove log** instead of **addlog** to the **vxassist** command. And as in the case of the mirror, if you also pass a number of logs, then this number does not specify the number of logs to be removed, but the number of logs to remain in the volume

!

Synopsis and sample command to remove a log from a volume

```
# vxassist -g <DGname> remove log <volname> [nlog=<x>] [logtype={dco|drl}
# vxassist -g adg remove log avol logtype=dco # will remove one DCO log
```

Growing a Volume with a File System

Growing a volume that has a **ufs** or **vxfs** file system on it requires two steps: first, the data container (i.e. the volume) must be resized so it can hold the desired larger amount of file system blocks. After this is done, the file system also needs to be resized to make use of the increased container size. It would not make a lot of sense to grow the container but have the file system still report the same old size. The superblock needs to be updated to reflect the new size, and in the case of **ufs**, new cylinder groups must be created and initialized. In the case of **vxfs**, the newly gained free space must be incorporated into the free extent list so that it can subsequently be allocated for files. There are convenient commands to grow a **ufs** or **vxfs** file system, so none of this has to be done manually.

The command to grow a volume that has a file system on it is one layer above **vxassist**. It is a tool that calls **vxassist** to resize the volume and then for **vxfs** calls **fsadm** (file system administration) or for **ufs** calls **/usr/lib/fs/ufs/mkfs -G** (the **-G** flag is undocumented) to grow the file system. The name of the tool is **vxresize** and it takes the volume name (and the ubiquitous **-g <DG>**) and a VxVM size specification. Alternatively, you can specify a size increment by prepending the number with a plus (+) sign. For **ufs** it does not make any difference whether the file system is mounted or not. You can do it offline as well as online. For **vxfs** however, the file system needs to be mounted. Why is that the case? Well, because of the rather simple file system structure and limited capabilities of **ufs** there exists a simple standalone command that can modify a **ufs**'s internal data structures without risk. On the other hand, the **vxfs** file system is highly sophisticated and complex. While in a **ufs** file system the file system metadata (like free block bitmap, inode table etc.) is static and its locations are fixed, **vxfs** allocates all file system metadata dynamically, as files (yes, **vxfs** metadata are files, too. Read all about it in the "point-in-time copies" (page 233) and "file system" (page 434) chapters). In order to allocate metadata for managing the extra volume space the file system driver must be used for that volume, so that (meta data) files can be modified. This means that the volume needs to be mounted.

Synopsis and sample command to grow a volume and its file system

```
# vxresize -g <DGname> <volname> <[+]size>
# vxresize -g adg avol +2g # enlarge avol plus file system BY 2g
# vxresize -g adg avol 20g # enlarge avol plus file system TO 20g
```

Growing a Volume Without a File System

Growing a volume without a file system, e.g. before putting a new FS onto it or if you are using database raw device access, can be done with a **vxassist** command. There are two subcommands to **vxassist** for growing: **growby** and **growto**, whose meanings should be immediately obvious. For **growby** you supply an increment, while for **growto** you supply the final size of the volume.

Notable fact: As soon as the command is executing it is safe to use the full new size of the volume. In the case of a redundant volume VxVM will still resynchronize the new extents, but that should not stop you from feeling comfortable using the new space. It is just as safe and redundant as the rest of the volume. If you find this hard to believe, look up RDWRBACK synchronisation in the index and read all about this interesting fact.

Synopsis and sample command to grow a volume without file system

```
# vxassist -g <DGname> grow[by|to] <volname> <size>
# vxassist -g adg growby avol +2g # enlarge avol BY 2g
# vxassist -g adg growto avol 30g # enlarge avol TO 30g
```

Shrinking a Volume With a File System

Shrinking a volume with a file system on it only works with **vxfs**, not with **ufs**. The action is in principle the same as for growing, but in reverse order and of course direction. First, the file system is told to shrink to the desired size using **fsadm -b**. Next, the volume is shrunk using the appropriate **vxassist shrinkto** or **vxassist shrinkby** command. This **vxassist** command must be run with the **-f** flag (force) because shrinking a container that contains a file system could result in loss of data. Of course vxresize is again the tool of choice, as it will do both in a single step with less fuss.

Synopsis and sample command to shrink a volume and its file system

```
# vxresize -g <DGname> <volname> <[-]size>
# vxresize -g adg avol -2g # shrink avol plus file system BY 2g
# vxresize -g adg avol 10g # shrink avol plus file system TO 10g
```

Shrinking a Volume Without a File System

Shrinking just the volume is easy: just issue the appropriate **vxassist shrinkto** or **vxassist shrinkby** command. You must use the **-f** option to force the operation, however. This is because all volumes created by **vxassist** are of usage type **fsgen**, which stands for **file system generic**. This is how VxVM finds out that it is meant for a file system and that it's not, for example, the root disk's **swap** volume which would not need to be resynchronised if re-opened after a crash etc.

In other words, VxVM will have to assume that you are using the volume for a file system and it wants you to think twice before you truncate the container that hold it. Of course, if you really know what you're doing then adding **-f** should not be a big extra effort. And if you really **don't** know what you're doing, then you are doomed anyway if they let you play with VxVM. No offense intended :)

Synopsis and sample command to shrink a volume without file system

```
# vxassist -g <DGname> shrink[by|to] <volname> <size>
# vxassist -g adg shrinkby avol 2g # shrink avol BY 2g
# vxassist -g adg shrinkto avol 10g # shrink avol TO 10g
```

Growing or Shrinking Just the File System

For **ufs** there is the command **/usr/sbin/growfs**, which is a shell script that calls **/usr/lib/fs/ufs/mkfs -G** to grow the file system. You can call the latter directly or use **growfs** as you like. Shrinking a ufs file system is not possible, but if necessary then you can first convert it to a vxfs using **/opt/VRTSvxfs/sbin/vxfsconvert**, linked to by **/opt/VRTS/bin/vxfsconvert**, and then shrink the vxfs file system in the way outlined below. The latter may not work well in older versions of VxVM because an important data structure for **vxfs** gets placed way back towards the end of the volume from where it unfortunately cannot be relocated. So it is sometimes not possible to shrink the volume significantly. It is probably the best to try it out with the software you are actually running.

For **vxfs** there is the tool **/opt/VRTSvxfs/sbin/fsadm**, linked to by **/opt/VRTS/bin/fsadm**, which can resize a **vxfs** file system in both directions: It is called with the **-b** flag and either the new size or the relative change as shown in the example below:

```
# vxassist make avol 1g
# mkfs -Fvxfs /dev/vx/rdsk/adg/avol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mkdir /vxfs
# mount /dev/vx/dsk/adg/avol /vxfs
mount: /dev/vx/dsk/adg/avol is not this fstype # Oops, need to specify vxfs!
# mount -F vxfs /dev/vx/dsk/adg/avol /vxfs # That's better!
# df -h /vxfs
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/avol   1.0G    17M   944M     2%    /vxfs # 1 Gig!
# vxresize avol +1g # Enlarges volume and then file system, too
# vxprint -v avol
TY NAME       ASSOC      KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
v  avol       fsgen      ENABLED  4194304  -        ACTIVE   -       -
```

Remember that VXVM sizes are always blocks if not otherwise given. So the above 4194304 blocks are exactly 2GB, not 4GB (Solaris disk blocks are 1/2KB each).

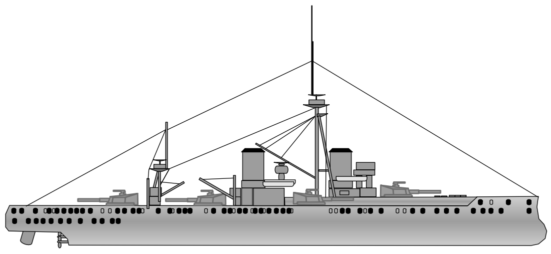
```
# df -h /vxfs
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/avol   2.0G    18M   1.9G     1%    /vxfs # 2 Gig!
# fsadm -b 1g /vxfs # We will have a 1G file system on a 2G volume!
UX:vxfs fsadm: INFO: V-3-23586: /dev/vx/rdsk/adg/avol is currently 4194304 sec-
tors - size will be reduced
# fsadm -b +1g /vxfs # Back to 2G! Just to try out all the ways...
UX:vxfs fsadm: INFO: V-3-25942: /dev/vx/rdsk/adg/avol size increased from
2097152 sectors to 4194304 sectors
# fsadm -b -512m /vxfs # Down to 1.5G!
UX:vxfs fsadm: INFO: V-3-23586: /dev/vx/rdsk/adg/avol is currently 4194304 sec-
tors - size will be reduced
```

```
# df -h /vxfs
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/avol   1.5G    17M   944M     2%    /vxfs # 1.5 Gig!
# vxprint -v avol
TY NAME       ASSOC      KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
v  avol       fsgen      ENABLED  4194304  -        ACTIVE   -       -
# vxassist -f shrinkto avol 1g+512m # Cool way of specifying volume size!!
# vxprint -v avol
TY NAME       ASSOC      KSTATE   LENGTH   PLOFFS   STATE    TUTIL0  PUTIL0
v  avol       fsgen      ENABLED  3145728  -        ACTIVE   -       -
# df -h /vxfs # File system is unchanged by vxassist shrink!
Filesystem             size   used  avail capacity  Mounted on
/dev/vx/dsk/adg/avol   1.5G    17M   1.4G     2%    /vxfs
```

Disadvantages of Using vxresize Over vxassist and fsadm

There are two slight disadvantages to using **vxresize** instead of issuing separate commands for **vxassist** and **fsadm** yourself: One is limited storage allocation. The **vxassist** command can be made to allocate very specific storage classes in a very fine-grained way. For instance, you can tell it to stripe across enclosures and mirror across controllers, to use only certain controllers or trays, or to exempt any of these objects and more. The **vxresize** command, while it ultimately calls **vxassist** itself, is not able to parse all of this storage allocation information. It is limited to a small subset of storage allocation. So while you can always use vxresize to shrink a volume you should use it for growing a volume only if you do not care where your subdisks will reside in the end. If you do, then it is better to use the combination of **vxassist** and **fsadm** instead.

If you wonder what **vxresize** does if the file system and volume sizes differ at the beginning, e.g. because someone messed with **vxassist** or **fsadm**, then the comforting answer is: it fixes the situation by making them both the same size. In other words, you can use **vxassist** to allocate storage with more control over the allocation. You can do so even if you forgot how to change the file system size, because instead of subsequently calling **fsadm** you can follow up with **vxresize**, which will then just adapt the file system if the volume already has the right size.


The Full Battleship

5.5 Tuning vxassist Behavior

It is sometimes desired to specify what parts of the storage equipment we want **vxassist** to use when allocating volume space. There are several keywords to **vxassist** that we can use on the command line to make **vxassist** do exactly what we want. In particular, you can control which disks or LUNs you want to use, which enclosures (storage arrays), which trays inside the storage array, and which controllers. You can also negate any of these criteria. You can specify whether to mirror across controllers, trays, or enclosures. And you can specify allocation of volume logs just like allocation of data plexes. The secret to this fine tuning lies in a few variables that can be passed to **vxassist** and that control its behavior. Let's go through them one at a time:

5.5.1 Storage Attributes – Specifying Allocation Strategies

Allocation can be modified by specifying the alloc=... variable on the vxassist command line. alloc= must be followed by a single shell-word containing a list of all entities which we want to allow **vxassist** to allocate from. The shell word can consist of just a single disk name, a comma-separated list of disk names, or a quoted string of space-separated disk names.

Synopsis and sample of a **vxassist** command with storage specification:

```
# vxassist make <volname> <size> [layout=<volume features>] [alloc=<disklist>]
# vxassist make avol 1g layout=stripe alloc=adg03,adg04,adg05
```

The latter can also be written without the "**alloc=**" keyword as shown below. This is a short form and you are free to use it, but it helps to remember that logically we are supplying an **allocation** information. Why does that help? Because later, when we **remove** objects instead of **creating** them we can also pass such allocation information. However, this will lead to those disks passed on the command line to be **excluded** from deletion, rather than **specified** for deletion. This is more understandable when you are aware that what you pass to **vxassist** is a list of storage objects where virtual objects are to **reside** on (i.e. allocated) rather than just a list of disks to "do something" with. That said, the following command line is equivalent to the previous one:

```
# vxassist make avol 1g layout=stripe adg03 adg04 adg05
```

Will specifying a storage allocation enforce VxVM to use all the specified disks? I.e. if you tell **vxassist** to create some small volume, but pass it ten LUNs as a storage allocation, will it then magically allocate one tenth of the storage from each of the LUNs passed on the command line?

No, it won't.

The storage allocation limits VxVM's normal allocation strategy to the given subset of all available storage. If the DG contains ten empty LUNs of ten GB each and you create a concat volume of 20GB, then VxVM will use exactly two LUNs out of the available ten to create the volume. If you give it a storage allocation of five disks, it will pick any two of the five that you passed. If you pass exactly two LUNs, it will of course pick those two. If you pass it only one LUN, the command will fail because not enough storage can be found in the storage allocation you gave.

Adding and Removing a Mirror on a Specific Disk

Just like you can control the storage allocation when creating a volume you can also control storage allocation when creating any other virtual object, e.g. another data plex, or a log plex (more about log plexes later). It works the same way. To add a mirror to a volume using specific disks you specify them on the vxassist command line:

Synopsis and samples of a **vxassist** command with storage specification:

```
# vxassist mirror <volname> [alloc=<disklist>]
# vxassist mirror avol alloc=adg00,adg01,adg02
# vxassist mirror avol adg00 adg01 adg02
```

Or you could specify to mirror across enclosures by this command:

```
# vxassist mirror avol mirror=enclr
```

To remove a mirror that resides on a certain disk you can also use storage allocation, but you need to revert the logic. Remember what we said above: Since it is an allocation that means we specify the LUNs that will **contain** objects after the command has finished. To remove a mirror from a specific disk, you need to use the negation operator "!" in front of the storage object.

Synopsis and samples of a **vxassist remove mirror** command with storage specification:

```
# vxassist remove mirror <volname> [alloc=!<disk>]
# vxassist remove mirror avol alloc=!adg00
# vxassist remove mirror avol !adg00
# vxassist remove mirror avol \!adg00 # If bash or a csh-variant is used
```

If you are using a csh-derived shell like **bash, zsh**, **tcsh**, or **csh**, then the command will most likely fail with a cryptic "**event not found**" error message. This is because these shells use the bang-operator ("!") to recall previous command lines, and unless you used a command line that started with the name of the disk (**adg00** in this case) the shell will complain. The solution is to Use A Real Shell™ (**/bin/ksh**) or to escape the bang operator with a backslash or use single- or double-quotes around the storage allocation.

Adding and Removing a Log on a Specific Disk

Of course you can also use storage allocation with any other object, for creation as well as for deletion (see above). For instance, logs can be put onto a specific LUN using the same approach when adding them after the volume has been already created.

Synopsis and samples of a **vxassist addlog** command with storage specification:

```
# vxassist addlog <volname> [alloc=<disklist>]
# vxassist addlog avol alloc=adg02
# vxassist addlog avol adg02
```

And again the same is true for removal of specific logs.

Synopsis and samples of a **vxassist remove log** command with storage specification:

```
# vxassist remove log <volname> [alloc=!<disklist>]
# vxassist remove log avol alloc=!adg02
# vxassist remove log avol !adg02
# vxassist remove log avol "!adg02"     # For bash users
```

There is also a possibility to specify storage allocation for data plexes and log plexes in the same command, but this makes use of the rather complicated suboption **-o ordered** of **vxassist.** It is rarely used and probably not very useful for you, so here is just one example that creates a mirrored volume with data plexes on **adg01** and **adg02**, plus a dirty region log on **adg04**:

```
# vxassist -o ordered make avol 1g layout=mirror,log alloc=adg01,adg02
logdisk=adg04
```

More Storage Allocation Variables

As we said before there are even more things that you can specify to direct storage allocation. You can set or exclude controllers, enclosures, trays, and targets. The following example shows how to mirror an existing volume **avol** using only LUNs that are attached to controllers c4 and c9, will exclude target c4t56 but specifically allow LUN c4t56d9:

```
# vxassist mirror avol ctlr:c4,c9 alloc=!c4t56,c4t56d9
```

To find out all about the possible criteria to allocation ask vxassist for an up-to-date explanation pertaining to the version you are running:

```
# vxassist help alloc
Allocation attributes for vxassist:
Usage: vxassist keyword operands ... [!]alloc-attr ...
   or: vxassist keyword operands ... wantalloc=mirror-attr[,attr[,...]]

 disk                   Specify the named disk in the disk group.
 dm:disk                Specify the named disk in the disk group.
 da:device              Specify a disk, by disk device (e.g., da:c0t2d0).
 ctlr:controller        Specify a controller (e.g., ctlr:c1).
 target:SCSI-target     Specify a SCSI target (e.g., target:c0t2).
 ctype:driver-type      Specify a disk controller type (e.g., ctype:emd).
 ctype:ssa              Specify SPARCserver Array controllers.
 driver:driver-type     Specify a disk driver type (e.g., driver:sd).

NOTE: wantalloc indicates desired, but not required, restrictions.
NOTE: !alloc-attr requests that the specified storage should NOT be used.
NOTE: for allocation attributes of the form attr:value, value can be „same“
      to indicate that allocations should use disks with the same value for
      the attribute (e.g., ctlr:same requests use of the same controller).
```

Specifying the Number of Data Plexes (Mirrors)

There is a simple parameter that you can pass to **vxassist** that lets you adjust the number of data plexes that our volume will have. The parameter is called **nmirror** and is used in a similar way to **alloc**, **layout** etc. You just specify the number to **vxassist.** For instance:

```
# vxassist make avol 1g layout=mirror nmirror=2
```

will create a 2-way mirror. A five-way mirror can be created by this command:

```
# vxassist make avol 1g layout=mirror nmirror=5
```

Up to 32 data plexes can reside in any volume. But remember that for reasons of both performance and redundancy no disk may ever be used for two different data plexes of the same volume, nor for two different columns of a stripe. So if you really were to create e.g. 10-way stripe across 5 columns, then your DG would have to have 10 times 5 equals free space available on fifty different disks. Otherwise you get the famous error message:

```
VxVM vxassist ERROR V-5-1-435 Cannot allocate space for <XXXXXXXXX> block volume
```

This error message usually comes from lack of independent disks rather than actual lack of **space**.

Specifying the Number of Log Plexes

The number of dirty region log (DRL) plexes or data change object (DCO) log volumes is defined using the **nlog** variable, just like the **nmirror** variable defines the number of data plexes:

```
# vxassist make avol 1g layout=mirror,log nmirror=4 nlog=3
```

Specifying the Number of Stripe Columns

The number of stripe columns is set using the **ncol** variable. If you do not explicitly set the number of columns then **vxassist** will calculate and use a default number based on the number of disks in the DG. If the volume is not mirrored then the number of columns will be at most half the number of disks in the DG (so the volume can later be mirrored). If it is mirrored then it will be calculated to fit onto the free disks in the DG. The actual number of columns will be between a minimum and a maximum value which you can inquire using the command **vxassist help showattrs**:

```
# vxassist help showattrs
#Attributes:
layout=nomirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-
col,nostripe-mirror-sd,
noconcat-mirror,nomirror-concat,span,nocontig,raid5log,noregionlog,diskalign,no
storage
mirrors=2 columns=0 regionlogs=1 raid5logs=1 dcmlogs=0 dcologs 0
autogrow=no destroy=no sync=no
min_columns=2 max_columns=8 # The default range for the "ncol" value
regionloglen=0 regionlogmaplen=0 raid5loglen=0 dcmloglen=0 logtype=region
stripe_stripeunitsize=128 raid5_stripeunitsize=32
stripe-mirror-col-trigger-pt=2097152 stripe-mirror-col-split-trigger-pt=2097152
usetype=fsgen diskgroup= comment="" fstype=
sal_user=
user=0 group=0 mode=0600
probe_granularity=2048
mirrorgroups (in the end)
alloc=
wantalloc=vendor:confine
mirror=
wantmirror=
mirrorconfine=
wantmirrorconfine=protection
stripe=
wantstripe=
tmpalloc=
```

Let us not get into too much detail about these variables. Suffice it to say you can change these defaults by entering them into the file **/etc/defaults/vxassist**.

Specifying the Stripesize

The size of a stripe unit, i.e. the number of blocks after which **vxio** jumps to the next column in a stripe, can be set using a large number of keywords. You can find them all by using the UNIX command **strings -a** on the **vxassist** executable and then pick the one you like best. Being a lazy UNIX person I prefer the shortest form, **stwid**:

```
# strings -a /opt/VRTS/bin/vxassist | egrep "^st.*wid"
stwid
stwidth
st_width
stripewidth
stripe_width
stripeunitwidth
stripe_stwid
stripe_stwidth
stripe_st_width
stripe_stripeunitwidth
```

So let's first find out what the default stripesize is, and then make a striped volume with a reasonably large stripesize (we discussed the doubtful merits of small stripesizes before)

```
# vxassist help showattrs | grep stripe_
stripe_stripeunitsize=128 raid5_stripeunitsize=32 # 128 blocks is way too small!
# vxassist make avol 1g layout=stripe ncol=6 stwid=2048 # 1MB is reasonable
```

Actually you can leave the **layout** keyword away in many cases. It's just a way of specifying some features with their default values. E.g. if you want to create a two-way mirrored stripe with one dirty region log and the default number of columns then you might specify it using this **vxassist** command line:

```
# vxassist make avol 1g layout=mirror,log,stripe
```

But if you need something special, like three-way mirroring, two logs, and four columns, then instead of writing

```
# vxassist make avol 1g layout=mirror,log,stripe nmirror=3 nlog=2 ncol=4
```

it is easier to just specify the individual features like this:

```
# vxassist make avol 1g nmirror=3 nlog=2 ncol=4
```

Likewise, if you want a volume with a striped layout with the default number of columns and a stripesize of 2048 blocks you could write:

```
# vxassist make avol 1g layout=stripe stwid=2048
```

or you could just make **vxassist** imply that you want a stripe by issuing this command:

```
# vxassist make avol 1g stwid=2048
```

Specifying a stripesize is sufficient because there is no other way for vxassist to satisfy your request for a certain stripesize but to create a striped layout.

5.5.2 Skipping Initial Mirror Synchronisation

When a volume is created manually then the first state that this volume will have is the **EMPTY** state. The **EMPTY** state means that VxVM has no idea about the validity of the data inside the plexes. They might be valid (e.g. if you were a very clever person) or they might be totally uninitialized (which is the more normal case). It may be a bad thing to use uninitialized data, so a volume that has an **EMPTY** state cannot be started without extra precautions. These precautions being to check if the volume is redundant. If it is not, it can indeed be started. But if it is indeed redundant, then all the plexes will first be synchronised so they all hold the same data. But while this precautionary measure is extremely safe it is not optimal, because it is usually unnecessary. We will learn both why it can be skipped and how to skip it a few lines from here.

The **vxassist** command will always try to give you a reliable, working volume. So **vxassist** will by default create the volume, then for redundant volumes start it by initiating a synchronisation process for the volume. The volume can be used immediately, but in the background a kernel thread will continue synchronisation until all of the volume is synchronised.

You can modify this behavior simply by passing a different initial state to the **vxassist** command. For instance, you can tell it to make the volume **ACTIVE** immediately, without synchronisation:

```
# vxassist make avol 1g nmirror=3 init=active
```

This will create a three-way mirror of 1GB without doing any synchronisation. You could also choose to zero out all data on the plexes, which is a faster way to start a new RAID_5 volume (because the parity of a stripe containing just zeros is also zero, the parity information need not be calculated if you initialize with zero) . The initial state to pass to **vxassist** in this case is **zero**:

```
# vxassist make avol 1g nmirror=3 init=zero
```

This will successfully create a three-way mirror of 1GB without doing any synchronisation. You can use **zero** initialization with all layout types. It is not limited to RAID-5 or mirrored volumes; you can even use it to initialize a concat volume.

Of course you can also specify that **vxassist** not initialize your volume at all, but leave it in an **EMPTY** state. In this case, you pass **none** instead of **zero** or **active** to **vxassist**:

```
# vxassist make avol 1g [nmirror=x] [ncol=y] init=none
```

There is much more on synchronisation mechanisms starting on page 380 in the troubleshooting chapter.

5.5.3 Changing the Layout of a Volume

Changing a volume's layout on the fly, while user I/O is active on it, is probably one of the coolest demonstrations that can be done with VxVM. Unfortunately many users do not trust the relayout feature and would rather not use it in production environments. Whatever their reasons may be – they might just not believe such a thing is possible – in all but the most exceptional cases relayouting a volume works really well. It can be interrupted (even by a system failure, e.g. panic or power loss) and will automatically restart upon volume start. It can even be stopped before it is done, and reversed to restore the original layout. All this is possible while the volume is active and undergoing user-I/O! But while the relayout engine in the back end is a truly remarkable feature (although parts of its implementation could be improved) the parameter processing for the **vxassist relayout** command is rather poorly implemented. For instance, even in versions of VxVM that used to search through all DGs to find the specified object, relayout was the one command that would not do it and instead required specification of the DG. Then, if your target layout features striping, the number of columns is not initialized from the default as it is when creating a volume. The **ncol** parameter appears to be uninitialized, so it is in many cases too large for the DG and the command fails. If the target layout features mirroring, the result will always be a layered volume even if you explicitly specified a non-layered layout. This is particularly ridiculous because the same command (**vxassist**) can then be used to convert the result to what you really wanted (a non-layered volume; you will learn more about layered and non-layered volumes soon). And while relayout can actually generate any kind of volume features it will refuse to work unless the internal layout of a plex actually changes. For instance, if you covert from a **concat** to a **mirror** and vice versa, relayout will complain that this is not a relayout operation (the mapping inside the plex does not change, but a plex is merely added or taken away). It will then try to **convert** the volume between layered and non-layered, which will also fail because this is not what was requested. So the command ultimately fails.

But if you relayout from concat to some other plex mapping, like **stripe** or RAID-5, and then use **vxassist relayout** to create your desired layout (mirrored concat), it will work just fine.

To sum it up: **vxassist relayout** is powerful, but very picky. But anyway, it is indeed a nice feature, especially when you finally learned to master it. So let's begin with a concat volume:

```
# vxassist make avol 1g [layout=concat]  # create our base volume
# vxassist relayout avol layout=stripe ncol=4 stwid=2048 # takes some time
```

(avol is now a striped volume!)

```
# vxassist relayout avol layout=stripe ncol=2 stwid=1024
```

(avol is now just 2 columns wide, with 1024 blocks stripesize)

```
# vxassist relayout avol layout=mirror,concat
```

(avol is now a mirrored layered volume; a concat-mirror)

```
# vxassist relayout avol layout=raid,nolog ncol=4 stwid=128
```

... and so on. Depending a little on the hardware it takes roughly one minute per GB of volume size, just like synchronizations and other low-level VxVM I/O. Such I/O is always throttled to pause for several milliseconds between I/Os in order not to overload the machine. There is actually no way to make that pause disappear. You can only increase it. You can also increase the size of the individual I/Os but in all the tests that we did over the years with several versions of VxVM it never did speed up the volume operations. Your mileage may vary.

VXRELAYOUT START / REVERSE / STATUS

While a relayout process is running you can inquire its status, pause it, restart it, or reverse it. And while it is running in reverse, you can do the same again, practically jumping to and fro between two layouts. Not that it makes a lot of sense, but it really is a cool demo, at least to those who still have to manually allocate partitions for use with Solstice Disk Suite (SunVM) or some Linux LVM.

Synopsis and example of the command syntax for handling relayout tasks:

```
# vxrelayout status <volname>
# vxrelayout status avol
# vxassist make avol 1g layout=concat
# vxprint -q -ht -gadg avol # -q to suppress headers, -ht for nicer output
v  avol         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     avol-01      adg01    0        2097152  0         c0t2d0   ENA
# vxassist relayout avol ncol=3 stwid=2048 &
[1]     21756
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   220         RELAYOUT/R 10.05% 0/4194304/419431 RELAYOUT avol adg
# vxrelayout status avol
 CONCAT -->  STRIPED,  columns=3,  stwidth=2048
 Relayout running,  15.00% completed.
# vxtask abort 220 # The relayout task can only be found using "vxtask list"
VxVM vxrelayout INFO V-5-1-2288 Aborting readloop (task 220)
VxVM vxrelayout INFO V-5-1-2291 Attempting to cleanup ...
VxVM vxassist ERROR V-5-1-2302 Cannot complete relayout operation
[1] +  Done(7)                 vxassist relayout avol ncol=3 stwid=2048 &
 # vxrelayout status avol
 CONCAT -->  STRIPED,  columns=3,  stwidth=2048
 Relayout stopped,  20.00% completed.
# vxrelayout start avol & # restart the relayout process
# vxrelayout status avol # and see how it's coming along
 CONCAT -->  STRIPED,  columns=3,  stwidth=2048
 Relayout running,  25.00% completed.
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   224         RELAYOUT/R 44.67% 0/3985408/1780288 RELAYOUT avol adg
# vxtask abort 224      # abort the poor relayout yet again
VxVM vxrelayout INFO V-5-1-2288 Aborting readloop (task 224)
VxVM vxrelayout INFO V-5-1-2291 Attempting to cleanup ...
[1] +  Done(7)                 vxrelayout start avol
# vxrelayout reverse avol # reverse the relayout process, go back to original
# vxrelayout status avol # and see how it's coming along
 STRIPED,  columns=3,  stwidth=2048 -->  CONCAT # going back to original layout
 Relayout running,  80.08% completed.
```

If you want to sleep really bad tonight, then you can try to parse the output of **vxprint -rt** for a volume that is currently undergoing a relayout operation. But be warned: it is not a pretty sight!

Technical Deep Dive

5.6 Methods of Synchronisation

Whenever there is more than one data plex (container for one instance of the volume's data) in a volume then there is at least a theoretical possibility that the contents of the plexes differ. They may differ for any several reasons, some of which are more, some less likely. Let's look at some of the more unlikely ones first:

- Bit rot, i.e. the unintentional flipping of bits on the medium, can be considered unlikely, although it does indeed happen occasionally. Disks are typically very well protected against this using Reed-Solomon code error checking.
- The probability for a bad data transfer for current disks is about 1x10-14 or one in 100,000 billion. This sounds like an extremely low probability, but let's see how quickly this would happen if you were to stream data from a disk 24 hours a day at a little over 60 MB/sec: 60 MB per second is 60 times 1024 times 1024 times 8 bits per second, or 503,316,480 bits. Let's round off all those numbers. 500,000,000 times the number of seconds per day (86400) is 43,200,000,000,000 or 4.32x10E13 bits/day. In other words it takes less than three days for an unrecoverable error to reach the host if the disk is on full blast. Fortunately for us while the error may not be recoverable but it still is detectable, so our host will just do a retry of the read request and everything should be fine. Similarly, if the disk writes a block and finds that it did not verify, then it will retry the write/verify cycle until the block has been correctly written, and/or it will revector the block to somewhere else.
- Someone using **dd** or so to write on the raw device that a subdisk resides on is also rather unlikely, although there are companies where this statement is debatable.

So if all those reasons are so unlikely, then what are the **likely** reasons for different contents in plexes of the same volume? The likely reasons are:

- The volume has not been initialized yet (this is very likely; it happens with every redundant volume we ever created).
- A new plex is just being added to the volume and therefore has not been initialized yet (this happens every time we add a mirror).
- A system failure has interrupted one or several writes to the volume (this happens whenever power is lost, the machine panics, or other rather low-level errors occur).

All of these cases are handled except for someone accessing the raw device below a subdisk. There is nothing that keeps anybody from doing this, and it's very hard to tell it happened. The first two (unlikely) cases are handled by hardware and OS mechanisms, the case of the raw device access using **dd** or some other tool would wreak havoc even if it was a partition and not a volume. The other three, the more regular cases, are handled very elegantly within the VxVM state machine.

There is one case left and that is if someone mounts a partition instead of a volume from an encapsulated and mirrored boot disk read-write. That is the one case where the administrator must be really careful about. That is and must remain a no-no until this chapter as well as the troubleshooting part is fully understood. Because you will learn how to modify VxVM's object states to make the VxVM state machine work for you.

Let us now return to the three cases that we admitted are relatively frequent: uninitialized volumes, uninitialized plexes, and plexes with open writes.

5.6.1 Atomic Copy

Whenever a new data plex is attached to a volume that already has a valid data plex, the new plex is first of all set to write-only mode. Being in write-only mode sounds utterly useless at first (after all, what would you do with data that you can write, but not read)? But in fact it is a very clever way to make sure that the volume can remain online while the plex is being synchronised. Being in write-only mode the new plex actually receives both synchronisation data from an existing plex and actual user-I/O. It is, from a writing standpoint, already active. If it was not active in write-only mode, then we would either have to stop I/O to the volume while the synchronisation is taking place. This would force us to take the volume offline. Or we would have to remember all the extents that have been written to since the synchronisation started, and resynchronize them later. But then while we are resynchronizing the changes, new changes may come in. We would be caught in a circle with an unknown end.

We will now start with an unmirrored concat volume and watch as a new plex is being attached. You can observe the write-only flag (**WO**) in the output of **vxprint**:

```
# vxassist make avol 1g # Create our well-known 1 GB concat volume
# vxprint -q -htv -g adg # Look at it: volume, plex, subdisk, nothing fancy
v  avol         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     avol-01      adg01    0        2097152  0         c0t2d0   ENA
# vxassist mirror avol & # Allocate a new plex and attach it in background
# vxprint -qhtvgadg # Look at the volume. See the WO mode to the right!
v  avol         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     avol-01      adg01    0        2097152  0         c0t2d0   ENA
pl avol-02      avol         ENABLED  TEMPRMSD 2097152  CONCAT    -        WO
sd adg02-01     avol-02      adg02    0        2097152  0         c0t3d0   ENA
# vxtask list # This command shows all VxVM kernel threads
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   160           ATCOPY/R 09.86% 0/2097152/206848 PLXATT avol avol-02 adg
```

Look at the output of the **vxtask list** command above. From left to right you see: the task ID, which you can use to **vxtask abort**, **vxtask pause**, or **vxtask resume** the thread. The next field would be the parent task ID, but since this is not a compound action consisting of several sub actions it is empty. Next is type and state of the thread. The type is interesting: it is shown as **ATCOPY**, with a state of **R**. We will get back to that type very soon, but let's jump ahead for now. The **R** stands for running, which is what the task is doing (the status **P** means paused, and **K** means killing the thread). The percentage is how much of the task has already been accomplished, and progress is basically the same, but given not in percent but as a tuple of "first block / last block / current block". The rest of the line is interesting because it actually tells us what action is happening: **PLXATT avol avol-02 adg** means that a plex is being **attached** to volume **avol**, the plex's name is **avol-02** and the activity happens in the disk group **adg.**

And what does **ATCOPY** stand for? It stands for atomic copy, because that is exactly what is going on here. The new plex is receiving the data in a single, atomic operation. If the process was stopped it would have to start from scratch again. An **ATCOPY** is a complete copy from one plex into one or several other plexes. This is what it looks like when you add two mirrors at once (to **bvol** in this case). You see that two plexes, bvol-02 and bvol-03, are being added in the same operation, i.e. the data is only read once and then written twice:

```
# vxassist -g bdg mirror bvol nmirror=2 &
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   170           ATCOPY/R 03.52% 0/2097152/73728 PLXATT bvol bvol-02 bvol-03 bdg
```

Coming back to our mirror operation on avol we see that the vxtask is done and that the volume has reached its final layout: two data plexes. And now both of them are read-write. In other words, after the plex has been fully initialized the WO flag is reset to RW (read-write) and thus the plex can start satisfying read requests, too, instead of being limited to just receiving synchronisation data plus all current write I/Os.

```
# vxprint -qhtvgadg
v  avol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl avol-01       avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01      avol-01      adg01    0        2097152  0         c0t2d0   ENA
pl avol-02       avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01      avol-02      adg02    0        2097152  0         c0t3d0   ENA
```

5.6.2 Read–Writeback, Schrödinger's Cat, and Quantum Physics

Atomic copy was really easy to understand, so let's switch to something intellectually more challenging. This may be one of the hardest parts to understand for anyone learning VxVM. It usually takes two or three attempts for anyone trying to understand it until "enlightenment" is reached, but it is worth the trouble, if just for the nice sizzling feeling when you finally begin to grasp the idea and the beauty of the concept. The train of thought required for its understanding is basically the same as that required for understanding the very basics of quantum physics: That the state of any object (especially a very tiny object like an elementary particle (electron, proton, photon, etc.) is undefined until somebody measures it! An object is in limbo until you look. This is a very unusual thought for most people. After all, we like to think that everything exists whether somebody is looking or not. But it turned out that it is just not so (in Physics), because it has been proven over and over again using hundreds of (rather complicated) experiments.

The theory of quantum physics sounded so wrong to many physicists in the early years of the twentieth century that a Mr. Schrödinger created a thought experiment for the purpose of demonstrating just how absurd the whole quantum theory was. We will need it for analogy later, so let's look at the thought experiment. it is called Schrödinger's cat paradox and it goes like this:

A cat (as a placeholder for any macroscopic object that you can touch and see directly) is put into a sealed container. The container also holds a technical apparatus that consists of

- A hammer
- A vial of poison (the poor cat is probably lucky it's just a thought experiment)
- An extremely tiny bit of radioactive material
- An amplifying device that will cause the hammer to hit the vial if a radioactive decay happens.

The amount of radioactive material is so small that the probability for a decay happening in one day is fifty percent. The theory (which, along with the theory of relativity, is now one of the best proven theories in physics) says the following:

If the decay happens then the hammer will hit the vial, break it, ad the cat will die from poisoning immediately

If no decay happens, then the cat will stay alive (assuming it has been fed etc.)

As long as we haven't measured if the decay has happened, the cat is: what?

Well: Quantum theory says that unless we have measured its actual state the cat is neither dead nor alive, but it is actually both, in an interleaved state. That is because the state of any elementary particle (like the particle sent off by the decay that triggers the deadly contraption) is undefined until it is actually measured. Therefore the state of the whole system internal to the box must be undefined. In other words the cat is both alive and dead at the same time, and only when we open the box will we find that it is still meowing or has actually been dead for some time. All the time before we looked the cat was both alive and dead.

Just in case that got you interested in the topic: There are many very good books on the topic and I am not qualified to try and summarize them; you'll just have to believe it for now, or get one of those books:

Now sit back and relax, free your mind and get ready to enter the world of **reliable uncertainty.**

Suppose we had a volume that contained more than one plex, i.e. it is a redundant, possibly multi-way mirrored volume. How is that volume written? It is written in the following way: If a write I/O is done for a file that was opened with the **O_SYNC** option then the write is completed after all plexes have been physically updated. That means the data has been persisted to all mirrors. Only then is control passed back to the caller. In this case, there is no uncertainty about the mirror contents, because the application only undertakes the next step in its processing when the write has completed to all mirrors. Keep this in mind: **O_SYNC** traffic is always written to all mirrors synchronously. There is therefore never a consistency problem with data that has been written with the **O_SYNC** flag set. All critical data, like file system meta data, database transaction logs etc. is written this way.

Resynchronizing Volumes

Now comes the hard part: Data that is written without the **O_SYNC** option is not flushed to all mirrors synchronously, but is flushed when the OS finds is suitable. For this reason, it is typical for an active mirror to have differing contents with respect to the most recent asynchronous write I/Os. So far, so good. Now our system crashes and reboots. Some questions immediately pop up:

1. Can the volume be used right away or do we need to synchronize first?
2. Where is the most current data?
3. How do we go about synchronizing the mirror contents?

The first question can be safely answered with "use it right away". Making software for highly available computers would not b very helpful if after a crash we stopped everything until we repaired our own data structures and kept the user volumes from doing useful work. So: all volumes are available immediately for read and write. This leads us to question 2, which takes the most effort to answer and to understand. I'll try to make it short: we do not care where the most current data is! There, it's out! What will you think of VxVM now that you know we don't care about your precious data? Is VxVM unreliable and useless?

It is most certainly not. Quite the opposite, actually. VxVM takes its job very seriously, and it never works less reliably than a simple partition would. But the fact is that its developers had an unusually clear view of exactly what is or is not required for a volume management to work right. You see, if VxVM was trying to save the most recent data, then it would make a lot of fuss and gain close to nothing, as we will see. Let's look at some alternative approaches that a normal programmer might implement:

Silly Approach #1

We always write to the first plex in the volume first, and only after data is persisted to that plex do we write the other plexes. This way, when the system comes back up after a crash, we know that this plex has the good data and we can use it to copy the good data over the stale data in all the other plexes. During that time the volume is only available read-only, because any updates would bring the plexes out of sync again. Once the process has finished, we can then enable the volume for read-write use.

This approach is often suggested when we ask Joe Sysadmin how they would think a volume management might tackle the problem. They never get it right, and they usually don't even understand the right answer, so don't feel bad if you don't get it the first time, OK? The problem with silly approach number 1 is that if we ask Joe Sysadmin what the volume management would have to do when a disk in the "good" plex has failed and this event has triggered a reboot (this is not a double fault so we need to catch it!), they fail to find an answer. Eventually they will say something like: Well in **that** case we'll just have to pick any one plex and use that as the source. To which my inevitable answer is "well, if picking any plex in **that** case is good enough, why shouldn't it be good enough in **any** case? The point being that if there are circumstances under which you need to fall back to a perceived "inferior" solution then it is better to get that solution right and use is exclusively than to have different ways of doing things depending on the circumstances. We might end up producing a lot of code paths that are hardly ever taken and contain bugs that are very hard to find.

So how do we get it right?

Silly Approach #2

We keep a log bitmap that tracks every I/O to every plex. Before a write I/O is flushed to a plex a bit in the log is set for the corresponding volume region that is written to. When the next plex is written the bitmap for that plex is set etc. When the system comes back online after a crash, we can just inspect the bitmaps to find which data plex holds the most current data.

True, but what effort are we going through? We need a synchronous write to a bitmap for every asynchronous I/O to every plex! This is outrageously expensive!

Well, says the silly developer, it may be expensive but isn't it worth the expense?

The very clear answer is: It is not. Look at what you gain: **nothing**! At least nothing worth mentioning. Remember that valuable data is written with the **O_SYNC** flag, and is persisted to all plexes before the write completes and returns to the user. That means that we are doing the logging, leaving a volume read-only, writing to a special plex first etc. just for the worthless asynchronous data! Our duty as a volume management product is simply to behave like a partition, and asynchronous writes to a partition are not guaranteed to persist either. Asynchronous writes are kept in the file system buffer cache until the OS decides to eventually flush them to disk. So trying to catch "the best" of those flushes is just utterly useless! Considering the overhead involved with any of the silly approaches their required CPU time and I/O capacity would be much better invested in just flushing the file system buffer cache more often!

Catching the most recent data on a mirrored volume is roughly equivalent to delaying the crash by some fractions of a second. What good is that? The valuable data is on disk anyway, and the asynchronous data is not valuable, so why save it?

VxVM's Approach

Having (hopefully) understood that it is worthless to try and save the "newest" data on a crashed mirror we arrive at question 3: How do we actually resynchronize a mirror?

This is a process that consists of several actions:

- Creating a "dirty region" bitmap in memory that marks those regions that need to be resynchronised (the "dirty" regions)
- Initializing the dirty region bitmap so that everything is marked dirty (i.e. needs resynchronisation)
- Starting the volume in a special Read-Writeback access mode (**RDWRBACK**)
- Spawning a kernel thread that reads the whole volume contents
- Resetting the access mode to normal once the thread has finished reading the volume

The secret to the resynchronisation process is the **RDWRBACK** mode. What happens in this mode?

Writing in **RDWRBACK** Mode – Nothing Special

In **RDWRBACK** mode a write is handled almost normally: data is written to all plexes. The difference is that if all plexes have confirmed that the data has been persisted to disk then the corresponding bit in the dirty region bitmap is reset.

Reading in **RDWRBACK** Mode – Very Special!

This is the interesting part: Remember that VxVM only has to make the volume look exactly like a partition just so the file system driver does not get confused and crash? Good! Now what does a partition **never** do that a mirrored volume **might** do? A partition never returns different contents for the same block when it is read more than once. That's the crucial part! That, and no more than that, is what VxVM must deliver! And how does it do it? It's actually very simple. Before a block (or region or extent) is read it is read the dirty region bitmap is inspected to see if that particular block of the volume is already in sync or not. If it is in sync, then the read proceeds normally. If the bitmap indicates the region is dirty, however, then the read is processed using the normal, round-robin pattern that is the default for all volume manager volumes. But then, **once the block is read its contents are not delivered to the user right away. Before the user sees it, the block is first copied to the appropriate location on all the other plexes.** Only after this copying is the block passed to the user process. Its contents are thus guaranteed never to vary between consecutive reads. The next read my indeed be satisfied by another plex, but that plex will contain exactly the same data, because that data has previously been copied to all the other plexes.

Of course, after the data has been copied to the other plexes the corresponding bits in the dirty region bitmap are reset to indicate that this particular portion need not be resynchronised over and over again.

This behavior is exactly analogous to Schrödinger's cat experiment. We do not know which version of the data we will be getting until we ask for it. Until then, all versions of the data – new, old, corrupt – are equally valid choices because they are exactly what could have resided on a partition after a crash, too. But once we do read a block, we commit the data that we read to be **THE ONE AND ONLY** data, and all other possibilities are immediately eliminated.

In principle we could live with this behavior forever. But it would be extra overhead having to check the dirty region bitmap on each read I/O just in case there still was some piece of the volume left out of sync. We could even crash and re-crash all the time while resynchronizing without changing anything in terms of volume reliability. We would just get the dirty region bitmap reset every time we crash, causing some extra I/O, but there would be no problem at all with data consistency.

But in order to make sure the resynchronisation process is finite so we can eventually get rid of checking the dirty region bitmap for every read I/O volume manager starts a kernel thread that reads the volume's dirty regions from beginning to end, throwing the results away. Because the volume is in **RDWRBACK** mode this means that all the dirty regions will have been read and their dirty region bits reset when the thread terminates and therefore the volume access mode can safely be reset to normal.

We know this is pretty tough stuff to understand, but we hope you made it. If you didn't get it the first time, sleep over it and try it again tomorrow. It's worth it for everyone who enjoys a beautiful software design.

5.7 Volume Features in Detail

5.7.1 concat

A volume of concat layout is much better than you may think. You will now learn why that is the case.

First of all, let's again look at the data transfer rate of a disk. A disk as well as a LUN in the year 2008/2009 transfers on the order of 50-100MB/sec. It does so very effectively when it is streaming the data sequentially across the channel. As soon as the head needs to move, however, we are again limited to about 100-200 transactions/sec. If those transactions were reads (which can not be buffered by the storage array's large cache), then we would accordingly be limited to let's say 200 reads per second. Say the size of the data to be read is 8KB, which it is in many databases. Then we are limited to 200 times 8KB, or1600KB or roughly 1.5MB/sec. This is about one percent of the sequential transfer rate. On top of that, non-sequential data transfers create a lot more work for the CPUs and the storage array front end controllers, which have to keep track of many more I/O requests than in the case of a large sequential I/O.

But: Do we have any chance to keep scattered reads from happening? We hardly ever do, as this is dictated by the application. We could try to influence the application developers, and they might even listen, but other than that, our I/O subsystem will just have to satisfy whatever request comes.

So why do we even read this chapter, if we cannot keep scattered reads from happening? Well, what we do have is a chance not to make things **worse**! We'll show you how to avoid this:

Never stripe volumes with a small stripesize across a large number of disks.
When in doubt, use concat instead.

A concat layout will not introduce additional CPU load and seek traffic to sequential I/Os. Striping does that by splitting a single I/O into several smaller ones, which then need to be serviced by the I/O subsystem. Read more about this in the next part.

5.7.2 stripe

Unlike you probably think striping will typically **not** improve random I/O. It will also tend to **slow down** sequential I/O. There is a myth about how striping makes volumes faster, but that myth is based on very old data from times when it was true. On today's hardware it will generally tend to make things worse instead of improving them because of a number of reasons. First of all, striping was invented for load balancing and for parallelizing I/O, which both sound like A Good Thing™. Even when striping was invented, there was noticeable extra CPU load and a noticeable increase in volume latency. But those negative effects were offset by the advantage of being able to do multiple I/Os in parallel, so the overall effect of striping was perceived as positive.

But due to Moore's law and the problem with mechanics the numbers shifted over time by factors between 10 and 20,000, so what we are left with today is just the increased latency and increased CPU load, while the effect of increased parallelity is negligible. Look at the following example:

Striping on Early Disk Drives

In the days of the old disks, when you did a sequential I/O to a single disk, the disk controller initially had to wait for the first sector to fly by, then it started reading and transferring sector by sector to the host (we'll leave out the possible interleaving factor for simplification). Reading the sectors may have taken a few revolutions, so let's say the I/O took the following times: half a rotation (on average) to wait for the initial sector, plus (say) four rotations for the data transfer, because there were relatively few sectors on each track. In total, this would be 4.5 rotations.

Doing the same I/O and a volume that was striped across eight disks would exhibit a totally different behavior: On the plus side, the number of rotations (once the right sector was under the read/write head) would sink by a factor of eight. So instead of four rotations there would only be half a rotation for the transfer. All I/Os was parallelized, and this was obviously faster than before.

On the minus side, however, there was an increased latency in waiting for the first sector. This was because now we did not have to wait for **one** first sector but for eight different ones. And the chance for one of those sectors being relatively far away from the read/write head was eight times higher than before. So the rotational latency would increase. Let us say for the sake of simplicity that the rotational latency increased from an average of 0.5 to an average of (close to) one. Then the whole transfer would take one rotation for the latency plus half a rotation for the parallel transfers, or 1.5 rotations total.

Obviously, that was a huge advantage.

And then Moore's Law came into play.

Striping on Current Disks and LUNs

The amount of data on a track multiplied by factors of several thousands, while due to the limitations of the mechanical systems on a disk the rotational speed merely doubled or tripled (from 3,600-5,000rpm to 7,200-15,000rpm). So what does a striped I/O look like today?

A data transfer that would be satisfied from a single disk (or concat volume) would take half a rotation (average) to wait for the first sector, and then only a few degrees, maybe one tenth of a rotation, for the data transfer. The sum is 0.6 rotations. Stripe this across eight disks and you double the average initial latency to almost one rotation, and then divide the tiny part that actually transfers data (one tenth of a rotation) by eight. The total is one rotation plus 1/80th or so, which is negligible.

Congratulations, you just increased the latency, loaded your CPU with seven extra I/O setups (each of which take about as long as a 64K block transfer), loaded the storage array's front-end and back-end controllers with extra I/Os, trashed the read-ahead cache in the storage array, and put extra seeks onto everybody else's LUNs (which use the same physical disks that your LUNs use). Your array vendor will gladly offer you an upgrade to a more expensive machine!

I think we agree that striping is not a good idea for sequentially accessed volumes.

How about random I/O?

If you think about it you will find that random I/O across the whole volume is not helped by striping at all. Random I/O, by definition, is distributed across a large volume area, and whether you stripe your volume or concat it does not make any difference in the distribution of I/Os that hit each disk. So striping brings no advantage for random I/O either. To sum it up, striping brings a disadvantage to sequential I/O, and no advantage to random I/O.

Your DBA may demand an 8KB stripe size for the database volumes because (as many DBAs think) this improves database performance by distribution of I/O requests across all LUNs. Such a DBA is probably not aware that random I/O is distributed across the storage anyway, and those nice large sequential I/Os will be hacked into minced meat by the time they reach the storage array

Having said that, there are still (a few) reasons pro striping. For instance, if your volume was not evenly filled with data but the data only occupied the first part of the volume, then it would indeed be better to stripe the volume in order to distribute some of the I/O to other physical disks. Keep in mind, however, that modern file systems already try to distribute their allocations across all the available space. The old **ufs** file system uses cylinder groups for that purpose, **vxfs** allocates across the whole volume also, but without the need for cylinder groups. If you are using data base raw device I/O that is one case where such a usage pattern would be conceivable.

Another reason may be controller or path saturation in the host or in the storage array, which could be alleviated by striping.

Whatever your reasons are to stripe your volumes, be aware of the following basic facts:

- The more columns a stripe has the more the volume's latency will increase
- The more columns a stripe has the more the probability for fault increases. If any column fails the whole volume is unusable.
- The smaller the stripe size the more extra I/Os you create because I/Os span more than one column and must be sliced into more than one physical I/O
- The worst case is a small stripe size and a misaligned data file on top. Imagine an 8KB stripe size and 8KB I/O happening on it, but at an offset of 4KB. Every single I/O will have to be sliced in two and handled separately.

If you do not have dedicated physical disks for yourself, then the more you stripe, the more you impact everybody else's performance.

5.7.3 MIRROR

Redundancy is not an option, it is a necessity. Data needs to be redundant in order to be reliably accessible. The main question is where to put the redundancy: in the storage array or in the host. If you use redundant LUNs then in many cases they will be some implementation of parity-stripe, i.e. RAID-4 or RAID-5. There is not much to say against this concept, as long as the implementation is done in hardware, with predictive error analysis and automatic reporting, and with a large enough write buffer. All of this is usually the case with the large storage vendors. In many cases you can also use mirrored LUNs internally, but this often is not much better than the internal parity stripes. If you use host based mirroring anyway, e.g. because you are mirroring between two or more data centers, then you may decide to use non-redundant LUNs. There is no Right Way™ to do it, so I will just give you some help for your decision for or against host-based mirroring.

Host–Based Mirroring vs. Storage–Based Mirroring

Storage-based redundancy has several advantages and few disadvantages, but please weigh the arguments yourself.

Arguments pro storage-based mirroring are:

- You only transfer that data once across the channel; the storage array's CPUs take care of creating the redundancy.
- Storage-based redundancy exposes you less to media errors. Basically, your disks never seem to fail.
- You needn't check your volumes for disabled plexes as frequently and thoroughly as you would with non-redundant storage.
- You don't generally need to deal with recovery from media errors in VxVM.
- VxVM won't need to deal with failures to write to Private Regions.

And the arguments pro host-based mirroring are:

- System administrators can manage redundancy according to demand. Mirrors can be added and removed, even low-level repairs can be done because the system administrator has access to all the basic data structures from VxVM. Of course, that requires quite a lot of expertise.
- Host-based mirroring can be cheaper in those cases where remote mirroring is employed. If your system mirrors data to a remote data center then using mirrored storage in both data centers would effectively constitute a four-way mirror, which costs a lot more than a simple, two-way mirror. While remote mirroring can be done inside the storage array, too, it typically has issues with latency and I do not generally advise it. You won't believe how slow the speed of light is when used with those synchronous replication/mirroring protocols! On the other hand, if one of your data centers fails and all you have is host-based mirroring, then be aware that for the whole time between failure and full resynchronisation you will have no redundancy at all. This may or may not be acceptable to your company.

To sum it up: storage-based mirroring helps everywhere and just hurts your wallet. Host-based mirroring can be a nice addition if you mirror between locations, which cannot usually be done efficiently by storage-based (remote) mirroring.

VxVM Mirror Read Policy

If you decided to use host-based mirroring then it is good to know what exactly VxVM does when it writes to or read from a mirrored volume. Here's what it does:

Write I/O: Asynchronous

If data is written to a mirror without the **O_SYNC** flag set (i.e. normal user file system I/O), then the write is scheduled for all plexes and control is immediately returned to the user process. The actual I/O operations are initiated as the queue is being processed, and they will typically complete out-of-sync. So if there are asynchronous I/Os on a mirror, then even if the user process that initiated the write has regained control it is not sure if the data has been persisted onto all plexes, onto some plexes, or even onto any plex. This sounds alarming, but it is in fact identical to the behavior of a partition. The SCSI driver will acknowledge an asynchronous I/O to a partition before the SCSI I/O has actually completed. Even if the user process has regained control (i.e. the system call has returned) it is not sure whether or not the data has been flushed to disk. This is not normally a problem because normal file I/O is not considered critical. If critical data are written, like database entries or file system meta data, then those data are written synchronously (see below).

Write I/O: Synchronous

A write that carries the **O_SYNC** flag or that is executed on a raw device or a mount point that has been mounted with the **-o directio** option will not return to the initiating process before all plexes have been successfully written to. I.e. if the user process regains control after writing synchronous data then it is guaranteed that all instances of the data on all plexes are identical. This is important because synchronous I/O is typically generated by applications that require some kind of guaranteed behavior. When a synchronous write returns and data has not actually been persisted then this breaks the software's writing paradigm and will eventually lead to unpredictable results.

Read I/O

A read from a mirrored volume is satisfied from any one of the active plexes, i.e. those plexes that contain valid data (those that have failed in the meantime are flagged accordingly and not used any more). There are two ways that VxVM uses to read from a mirror: One is called "Round Robin" (**rdpol=round**). This means that reads are satisfied from one plex after the other until all plexes have been used, and then the first plex is used again. This is done in order to balance the load between the individual LUNs or disks. The other one is called "Preferred Plex", and the preferred plex is named and associated with the volume. This means that all reads are satisfied from that preferred plex (because it has faster storage, because that storage is located closer to the host and thus has lower latency, etc.).

By default, volumes have a read-policy called "Select", which means no more than VxVM will select the most appropriate read policy between "Round Robin" and "Preferred Plex" by examining the volume layout. If the volume consists of plexes with identical layout, then the "Round Robin" policy is used. If one plex has a larger number of stripe columns than the others then that the read policy will be "Preferred Plex" for the plex with the highest number of columns. To set the read policy to "Preferred Plex" manually, or back to "Round Robin" or "Select", use the **vxvol rdpol** command:

```
# vxvol rdpol prefer avol avol-02
# vxvol rdpol round avol
# vxvol rdpol select avol
```

5.7.4 RAID-4 and RAID-5

These RAID-Levels sound good, because they combine the classic stripe load distribution scheme with volume redundancy. But that comes at a price. In general, RAID-4 and RAID-5 stripes suffer on small writes. You will soon see why. Their redundancy is also severely limited, and in summary, using software implementations of RAID-4 or RAID-5 for enterprise systems does not usually make much sense. Veritas Volume Manager does not implement RAID-4, but RAID-5 is offered (see The Full Battleship). Since this is the Technical Deep Dive section, let's look at how RAID-4 and RAID-5 are implemented. We will need to look at four major areas: Parity calculation and distribution, read/write behavior, degraded mode (i.e. after a single media has failed) and recovery behavior.

Parity Calculation

Parity calculation is a very clever scheme to quickly recover lost data. This is how the principle works: Take any number of bits, i.e. ones and zeros, and note if the amount of ones in that set of bits is an even or an uneven number. E.g. the bit pattern

```
1 0 1 0 1 1 0 1
```

consists of three zeros and five ones. Five is an uneven number so we note that the parity of this bit pattern is uneven. We do this by setting its parity bit to one. The resulting extended bit pattern (the bit pattern including the parity bit) will thus have even parity.

1 0 1 0 1 1 0 1 **1**

Now if any of the bits in the original pattern were to get lost we could use the parity information to calculate what the original bit had been by just doing the parity check again. If the parity of the current bit set (including the parity bit, but not including the bit that was lost – we cannot access it) is even, then the original bit must have been zero. If the parity is uneven, then the original bit must have been one. Of course, if we lost another bit, then all would be lost and we would have no chance of recovering the lost bits.

This parity scheme is how RAID-4 and RAID-5, which are therefore called parity stripes, are implemented. The individual bits in out bit set represent the columns of the parity stripes. The parity bit represents the additional parity column of the parity stripe volume. Of course addressing and counting every single bit in every block of the stripe on every write would be extremely slow. Instead of actually counting the individual bits, the CPU's **XOR** operation is used, which can be applied to whole words rather than bits. The **XOR** operation combines two values bitwise such that the resulting bit is equal to one if exactly one of the input values was equal to one. If they are both zero, the resulting bit is zero, and if they are both one, the resulting bit is also zero. In other words, **XOR** flags differences in the input values. In effect, this yields exactly the parity information. So what happens in a parity stripe is that the blocks on each stripe are combined using blockwise **XOR** and the resulting block, that contains the parity information for the whole stripe, is then written to the separate column in the RAID-4 implementation.

It quickly becomes obvious when one thinks about the problem for a while that the parity column will be overloaded with I/O as soon as several processes write to the RAID-4 volume. Each write needs to flush the corresponding parity information to the dedicated parity column, so while user data my be distributed to different disks, each write also puts load on the parity column. This is why in the next iteration, RAID-5, the parity was no longer put onto a dedicated column, but rather distributed across all columns. The technical term for the distribution, in case you are interested in obscure and useless tidbits, is "left-symmetric layout".

Read Behavior

Read behavior for RAID-4 is identical to a common stripe, read behavior for RAID-5 is a little bit different because the parity information is distributed and thus one more column is effectively working for user I/O. Hence, RAID-5 read performance tends to be a little better than a standard stripe, because it uses one column more than a normal stripe of the same size.

Write Behavior

Write behavior is indeed very interesting for parity stripes. We are rather sure that after reading this, you will quickly forget about parity stripes and use different layouts from then on.

Full–Stripe Write

In the best case, a full stripe is written. This is a relatively quick and easy process: The parity information is calculated from all the data buffers, the data is flushed onto the data columns for that volume region, and the parity information is written onto the parity column for that volume region (remember that the parity column is not constant with RAID-5). This is a little work for the CPU as well as an extra I/O for the parity column, but one extra I/O is certainly less than twice the I/Os, as we would have to do if the volume was a mirrored one. However, consider what happens when the write is interrupted because of a system fault (panic, power loss etc.). The parity information would not match the data any more. If you imagine a case where the system panics because of an error in the SAN that makes the system lose a disk in a funny way, then VxVM would have to reconstruct the data on the missing column using outdated parity information. If this happens to an area that holds important meta data then the meta data that is calculated by **XOR**ing the remaining columns with the parity column software will be seriously corrupt. It will be random data. Not just old data, but random! You don't want random data on your superblock, do you?

How do we solve the problem? We solve it by adding another protection mechanism called a transaction log. The transaction log is located on a different LUN from the data and parity column. It represents a circular buffer that holds (in the case of VxVM) the last five writes to the parity stripe. That means that all write I/O is effectively done twice: first to the transaction log, then to the actual data and parity columns.

Storage arrays will store the transaction log in non-volatile memory instead of on disk, which makes such hardware implementations much faster. They can also use special hardware for parity calculation, which again speeds up the hardware implementations vs software implementations.

In any case, storage array or VxVM, if the volume is started after a crash, all transactions from the transaction log are reapplied to the parity stripe in the order they appear in the log, in order to ensure consistency between data and parity and also to have the most current data available.

Most–Stripe Write

If most, but not all columns are written then if we just write the data the parity will be out of sync with the data because the new data is not represented in the parity. If we write data plus the newly calculated parity of the data the problem is not solved because then the remaining old data is not represented in the parity. What we need to do is actually read the remaining old data columns, the ones that will not be overwritten, then calculate the parity using that old data plus the new data that is to be written, and then write the new data and new parity. Of course, since we write via the transaction log, what actually happens if we do a most-stripe write is this:

1) Read remaining columns
2) Calculate new parity
3) Write new data plus new parity to transaction log
4) Write new data plus new parity to their respective columns

Doesn't sound quick, does it? Now look what we have to do when we write just a small number of columns

Few–Stripe Write

In this case it would be suboptimal to read all the remaining columns from the volume; there could be many. Instead, we read the columns that we are going to overwrite, plus the parity column. Then, we **XOR** the old data out of the parity data, **XOR** the new data into the parity, write the transaction log and finally the volume. It looks like this:

1) Read columns that will be overwritten
2) Read parity column
3) XOR old data with parity to extract old data's parity
4) XOR new data with parity to insert new data's parity
5) Write new data plus new parity to transaction log
6) Write new data plus new parity to their respective columns

Doesn't sound efficient either, does it? Wait until you see degraded mode!

Degraded Mode Read/Write Behavior

When one of the columns in a parity stripe fails the volume is switched to degraded mode. This means several things:

- The volume is no longer redundant
- Reads from existing data columns are satisfied in the normal way
- Writes to the volume proceed in the normal way
- Reads from the failed data columns are satisfied by reconstructing the missing data.

Reconstruct-read, as it is called, is performed by using all the other columns plus the parity information to calculate what had originally been on the column that is no longer accessible. What does that mean for the volume`s performance and reliability? It means that when, for instance, in a 10-column RAID-5 volume one disk fails, then for 10% of the read I/Os (the ones that would read from to the missing disk), nine columns must be read instead of just one, i.e. the number of disk accesses caused by reads practically doubles. Additionally, the **XOR** calculations must be performed by the CPU. Imagine a RAID-5 volume that is suffering from lots of scattered reads (the worst case today, as we proved near the beginning of this book). If one of the disks overheats, e.g. because a fan in the tray is broken, then all the other disks will have to deliver twice the amount of reads! This will not only slow down the performance even more, but it will also lead to more heat, increasing the probability for another failure. But now we are no longer redundant, so if the next disk fails the volume will become inaccessible!

Recovery Behavior

As soon as a column has failed in a parity stripe we must restore redundancy in order to keep up reliability. Unfortunately, what that means is that the replacement for the failed disk must be initialized with exactly the data that was lost. No problem in principle, since we can always reconstruct that data using the parity column plus all the remaining data columns. But if you read the last paragraph then you know that by now the number of read-I/Os to the remaining disks has already doubled, and now we're forcing even more I/O onto the system by systematically requesting all the data in the lost column – even the empty blocks since we are reconstructing on the raw device level! Keep in mind that for every extent whose data we want to reconstruct we have to read the corresponding extents on all the other data columns plus the parity column, and then calculate the original data and write it to the extent! This is an enormous strain on the I/O subsystem as well as the CPU. It therefore takes a long time, during which we are very slow and very vulnerable to further failures.

It is for these reasons that I do not recommend using parity stripes for enterprise applications; at least not software implementations of parity stripes. Hardware implementations are better because they do not require physical writes to a transaction log (which is in battery-backed memory), they do predictive failure analysis pretty well and they usually have hardware-assisted parity calculation engines to reduce the CPU load

5.7.5 mirror-concat

This is actually exactly the same as a plain mirror. Concatenation is the default layout, and even a volume with just a single, tiny subdisk is called a concat volume. This is similar to the **cat** command in UNIX, which is called that way because it **can** concatenate several files, but you still **cat** a file even if it's just a single one.

5.7.6 mirror-stripe

A mirror-stripe layout combines striping with mirroring. It is equivalent to the combined RAID level RAID-01. The basic plex layout of the volume is stripe, and another striped plex is attached to the volume. This has the advantage of adding mirror-like redundancy to the stripe, which is much better than a parity-stripe. If any of the disks in plex 1 fails then plex 2 still has the complete set of data. Further disk failures in the plex 1 stripe do not affect the volume because plex 1 is detached as soon as the first failure is detected. If another disk failure occurs in plex 2, the volume will be unavailable because plex 2 is also detached due to the disk failure.

Read/write I/O is performed according to the same read policy as a mirror (round robin/select/preferred plex), and of course data is always striped across all columns as it is in a stripe.

There is a much better layout called stripe-mirror, or RAID-10, which instead of mirroring stripes will create individual mirrors and then stripe across them. This is called a layered volume and is covered in the chapter on layered volumes.

5.7.7 Mixed Layouts

It is possible in all UNIX versions of VxVM to have a different layout in every plex. It is not possible in the Windows implementations, but that operating system has more serious limitations to worry about than VxVM limits anyway.

You cannot create mixed layout volumes with a single vxassist command, but once you have a volume you can just add plexes to it using vxassist mirror, and specify any kind of layout you want. Actually, adding a RAID-5 plex to a volume or adding another plex to a RAID-5 volume is not a straightforward task with vxassist either (it requires using layered volumes, but it does work). Once you have the multi-layout volume you can use it just like any other volume, but some features will not work any more. These are:

- Snapshots
- Fast mirror resynchronisation
- Relayout
- Resize

For these reasons you may want to stick with standard layouts. They are more thoroughly supported. But it sure is good to know that there is no hard limit inside VxVM that enforces identical layouts for all data plexes.

5.8 Relayout in Detail

The relayout feature of VxVM is pretty fascinating to watch in action, and you may wonder what is going on inside. We can tell you some of it, but we did not write the code, and the actual behavior varies with source and destination layouts, so please pardon if sometimes the details may seem a little odd. In order to understand this explanation it is absolutely necessary that you have a good understanding of what a layered volume is, so please make sure you have read the chapter on layering and layered volumes.

What happens when you relayout a volume is this:

- First of all the volume to relayout is being layered and pushed down several layers (depending on the exact parameters, typically to layers three or four). This allows VxVM to still access the volume from both user and kernel perspective, while at the same time enabling very thorough rearrangement of plexes and subdisks inside the volume.
- The next thing that **vxassist** does is look at the size of the volume to find the right size for a mirrored internal temporary buffer subvolume that the relayout process uses to copy data from the source subvolume to the destination subvolume. This buffer is then created as just a normal, mirrored volume, which is then layered and stuffed deep down into the volume that is about to undergo relayout. We will find usually it in layer three or four, and if you look at the output of **vxtask list** right after you start the relayout process then you will find a subvolume that is being synchronised using the **RDWRBACK** synchronisation method. This is the synchronisation of the internal buffer subvolume.
- If the volume to relayout is very small (i.e. smaller than 50MB) then it creates a buffer subvolume that is the same size as the volume. If the volume is between 50MB and 1GB then it will create a 50MB buffer subvolume, and if the volume is larger than 1GB it will create a 1GB buffer subvolume. Actually the 50MB value is outdated now, but it used to be true. With Storage Foundation 5.0 the value seems to be more dynamically allocated and generally be slightly more than 10% of the source volume's size.
- The next thing that happens – after persisting the intended relayout operation in the private region – is that the relayout kernel thread fills the buffer volume with data from the beginning of the source subvolume. Let us assume the extent of the buffer subvolume is 50MB. It then maps the buffer subvolume into the first 50MB of the source subvolume and unmaps the first 50MB of the source subvolume to free the subdisks. Now it allocates and maps the first 50MB of the destination subvolume in the correct destination layout and begins to copy the data from the buffer to the destination subvolume. Remember that all those subvolumes are contained in what the user uninterruptedly sees as the original data volume. They are just pushed way down several layers.
- Once the buffer subvolume's contents have been copied the first 50MB of the user volume are remapped to point to the new, destination volume. So from now on accesses to the first 50MB extent will be directed to the target subvolume, and accesses to the rest will be directed to the source volume.
- Now the circle repeats with the next 50MB, then the next, and so on, until the whole

source subvolume is copied onto the target subvolume.

- Then, the buffer subvolume as well as the rest of the source subvolume are freed, the target subvolume is unlayered and the relayout intent removed from the Private Region. Relayout is finished.

What actually happens is much more complicated than that. You will see up to seven subvolumes involved in a relayout process, not all of which can be easily explained, but this is the rough idea of it.

Since we are in the Technical Deep Dive section anyway, here is a short walk-through of a relayout:

```
# vxprint -qrtL -g adg # -q: no headers, -rt: more info, -L: separate layers
v  avol         -            ENABLED  ACTIVE   4194304  SELECT    -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   4194304  CONCAT    -        RW
sd adg01-01     avol-01      adg01    0        4194304  0         c0t2d0   ENA
# vxassist relayout avol  ncol=4 & # Start the relayout process
[1]     21976
# vxtask list # Let's look at the temporary buffer volume
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   241         RDWRBACK/R 16.12% 0/419328/67584 VOLSTART avol-T01 adg
# bc -l # Let's see how large the buffer subvolume "avol-T01" is
419328/2048
204.7500000000  # 204.75 MB, around 10% of the volume size
# vxtask list # Let's see if the actual relayout has started yet.
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   243         RELAYOUT/R 00.05% 0/8388608/4096 RELAYOUT avol adg
# vxrelayout status avol
 CONCAT -->  STRIPED,  columns=4,  stwidth=128
 Relayout running,  0.00% completed.
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   243         RELAYOUT/R 02.25% 0/8388608/188416 RELAYOUT avol adg
# vxprint -qrtL -g adg # Look at those six subvolumes in layers 2 and 3!
[...]
v  avol         -            ENABLED  ACTIVE   4194304  SELECT    -        fsgen
pl avol-tp01    avol         ENABLED  ACTIVE   4194304  CONCAT    -        RW

sv avol-ts01    avol-tp01    avol-I01 2        4194304  0         3/5      ENA
v2 avol-I01     -            ENABLED  ACTIVE   4194304  ROUND     -
relayout
p2 avol-Ip01    avol-I01     ENABLED(SPARSE) SRC 4194304 CONCAT   -        RW

sv avol-Is01    avol-Ip01    avol-S01 1        2936320  1257984   1/1      ENA
v3 avol-S01     -            ENABLED  ACTIVE   2936320  SELECT    -        fsgen
p3 avol-01      avol-S01     ENABLED  ACTIVE   2936320  CONCAT    -        RW
s3 adg01-01     avol-01      adg01    1257984  2936320  0         c0t2d0   ENA
p2 avol-Ip02    avol-I01     ENABLED(SPARSE) TMP 1677312 CONCAT   -        WO
```

```
sv avol-Is02    avol-Ip02    avol-T01 1        419328   1257984   2/2      ENA
v3 avol-T01     -            ENABLED  ACTIVE   419328   SELECT    -        fsgen
p3 avol-T01-01  avol-T01     ENABLED  ACTIVE   419328   CONCAT    -        RW
s3 adg02-01     avol-T01-01  adg02    0        419328   0         c0t3d0   ENA
p3 avol-T01-02  avol-T01     ENABLED  ACTIVE   419328   CONCAT    -        RW
s3 adg03-01     avol-T01-02  adg03    0        419328   0         c0t4d0   ENA
p2 avol-Ip03    avol-I01     DISABLED UNUSED   4194304  CONCAT    -        RW

sv avol-Is03    avol-Ip03    avol-U01 1        4194304  0         0/1      DIS
v3 avol-U01     -            DISABLED EMPTY    4194304  SELECT    -        fsgen
p3 avol-Up01    avol-U01     DISABLED(SPARSE) ACTIVE 5031552 STRIPE 4/128  RW
s3 adg01-02     avol-Up01    adg01    314496   943488   0/314496  c0t2d0   ENA
s3 adg04-01     avol-Up01    adg04    314496   734080   1/314496  c0t10d0  ENA
s3 adg05-01     avol-Up01    adg05    314496   734080   2/314496  c0t11d0  ENA
s3 adg06-01     avol-Up01    adg06    314496   734080   3/314496  c0t12d0  ENA
p2 avol-Ip04    avol-I01     DISABLED(SPARSE) WOD 0     CONCAT    -        WO

sv avol-Is04    avol-Ip04    avol-W01 0        0        0         1/1      ENA
v3 avol-W01     -            ENABLED  ACTIVE   0        SELECT    -        fsgen
p3 avol-Wp01    avol-W01     ENABLED  ACTIVE   0        STRIPE    4/128    RW
p2 avol-Ip05    avol-I01     ENABLED(SPARSE) DST 1257984 CONCAT   -        RW

sv avol-Is05    avol-Ip05    avol-D01 1        1257984  0         1/1      ENA
v3 avol-D01     -            ENABLED  ACTIVE   1257984  SELECT    avol-Dp01
fsgen
p3 avol-Dp01    avol-D01     ENABLED  ACTIVE   1257984  STRIPE    4/128    RW
s3 adg01-03     avol-Dp01    adg01    0        314496   0/0       c0t2d0   ENA
s3 adg04-02     avol-Dp01    adg04    0        314496   1/0       c0t10d0  ENA
s3 adg05-02     avol-Dp01    adg05    0        314496   2/0       c0t11d0  ENA
s3 adg06-02     avol-Dp01    adg06    0        314496   3/0       c0t12d0  ENA
```

We'll let some time pass here to give the relayout process some time to proceed with doing its work. Look at the highlighted output. What you can identify are the source subvolume (SRC), a temporary subvolume (TMP), an unused subvolume (UNUSED), a write-only and write-on-demand subvolume (WO, WOD), and the destination subvolume (DST).

```
# vxrelayout status avol
 CONCAT -->  STRIPED,  columns=4,  stwidth=128
 Relayout running,  69.98% completed.
# vxprint -qrtLgadg # Subvolume sizes have changed!
[...]
v  avol         -            ENABLED  ACTIVE   4194304  SELECT    -        fsgen
pl avol-tp01    avol         ENABLED  ACTIVE   4194304  CONCAT    -        RW

sv avol-ts01    avol-tp01    avol-I01 2        4194304  0         4/5      ENA
v2 avol-I01     -            ENABLED  ACTIVE   4194304  ROUND     -
```

```
relayout
p2 avol-Ip01    avol-I01     ENABLED(SPARSE) SRC 4194304 CONCAT   -        RW

sv avol-Is01    avol-Ip01    avol-S01 1        839680   3354624   1/1      ENA
v3 avol-S01     -            ENABLED  ACTIVE   839680   SELECT    -        fsgen
p3 avol-01      avol-S01     ENABLED  ACTIVE   839680   CONCAT    -        RW
s3 adg01-01     avol-01      adg01    3354624  839680   0         c0t2d0   ENA
p2 avol-Ip02    avol-I01     ENABLED(SPARSE) TMP 3354624 CONCAT   -        RW

sv avol-Is02    avol-Ip02    avol-T01 1        419328   2935296   2/2      ENA
v3 avol-T01     -            ENABLED  ACTIVE   419328   SELECT    -        fsgen
p3 avol-T01-01  avol-T01     ENABLED  ACTIVE   419328   CONCAT    -        RW
s3 adg02-01     avol-T01-01  adg02    0        419328   0         c0t3d0   ENA
p3 avol-T01-02  avol-T01     ENABLED  ACTIVE   419328   CONCAT    -        RW
s3 adg03-01     avol-T01-02  adg03    0        419328   0         c0t4d0   ENA
p2 avol-Ip03    avol-I01     DISABLED UNUSED   4194304  CONCAT    -        RW

sv avol-Is03    avol-Ip03    avol-U01 1        4194304  0         0/1      DIS
v3 avol-U01     -            DISABLED EMPTY    4194304  SELECT    -        fsgen
p3 avol-Up01    avol-U01     DISABLED(SPARSE) ACTIVE 13418112 STRIPE 4/128 RW
s3 adg01-02     avol-Up01    adg01    838656   2515968  0/838656  c0t2d0   ENA
s3 adg04-01     avol-Up01    adg04    838656   209920   1/838656  c0t10d0  ENA
s3 adg05-01     avol-Up01    adg05    838656   209920   2/838656  c0t11d0  ENA
s3 adg06-01     avol-Up01    adg06    838656   209920   3/838656  c0t12d0  ENA
p2 avol-Ip04    avol-I01     ENABLED(SPARSE) WOD 3354624 CONCAT   -        WO

sv avol-Is04    avol-Ip04    avol-W01 1        419328   2935296   1/1      ENA
v3 avol-W01     -            ENABLED  ACTIVE   419328   SELECT    avol-Wp01
fsgen
p3 avol-Wp01    avol-W01     ENABLED  ACTIVE   419328   STRIPE    4/128    RW
s3 adg01-05     avol-Wp01    adg01    733824   104832   0/0       c0t2d0   ENA
s3 adg04-03     avol-Wp01    adg04    733824   104832   1/0       c0t10d0  ENA
s3 adg05-03     avol-Wp01    adg05    733824   104832   2/0       c0t11d0  ENA
s3 adg06-03     avol-Wp01    adg06    733824   104832   3/0       c0t12d0  ENA
p2 avol-Ip05    avol-I01     ENABLED(SPARSE) DST 2935296 CONCAT   -        RW

sv avol-Is05    avol-Ip05    avol-D01 1        2935296  0         1/1      ENA
v3 avol-D01     -            ENABLED  ACTIVE   2935296  SELECT    avol-Dp01
fsgen
p3 avol-Dp01    avol-D01     ENABLED  ACTIVE   2935296  STRIPE    4/128    RW
s3 adg01-03     avol-Dp01    adg01    0        733824   0/0       c0t2d0   ENA
s3 adg04-02     avol-Dp01    adg04    0        733824   1/0       c0t10d0  ENA
s3 adg05-02     avol-Dp01    adg05    0        733824   2/0       c0t11d0  ENA
s3 adg06-02     avol-Dp01    adg06    0        733824   3/0       c0t12d0  ENA
[1] +  Done                    vxassist relayout avol  ncol=4 &
# vxprint -qrtLgadg
[...]
```

```
v  avol          -             ENABLED  ACTIVE   4194304  SELECT    avol-01  fsgen
pl avol-01       avol          ENABLED  ACTIVE   4194304  STRIPE    4/128    RW
sd adg01-03      avol-01       adg01    0        1048576  0/0       c0t2d0   ENA
sd adg04-02      avol-01       adg04    0        1048576  1/0       c0t10d0  ENA
sd adg05-02      avol-01       adg05    0        1048576  2/0       c0t11d0  ENA
sd adg06-02      avol-01       adg06    0        1048576  3/0       c0t12d0  ENA
```

The relayout has successfully completed, and the volume now has the desired target layout of stripe with four columns.

We hope this has enlightened you about several things:

1) You can trust VxVM's relayout feature. It is not magic, but actually understandable with reasonable effort.
2) Relayout would be really hard to do manually, if you had to allocate your own storage objects.
3) It actually works online, with no downtime to the application.
4) It is crash-proof and can be restarted and reversed at will.