CHAPTER 7: LOGS

by Volker Herminghaus

7.1 Overview

Using Veritas Volume Manager will make your data accessible, fast, and reliable. But as we said before there is nothing that cannot be improved. Consider the case where a mirrored volume is open while a system crash occurs. Because the plexes may not have been updated at identical times (this is optimistic - it would be a great coincidence if they actually **had** been updated at the same time) they probably contain different data in at least some volume regions. This would break the definition of volume management behavior: that a volume shall behave exactly the same as a partition. While a partition will always deliver the same data for the same blocks unless you change the blocks, a mirror with non-synchronous contents may not. This could cause the file system or database to crash, resulting in system panic or database corruption.

As another example of optimization potential consider a plex in a mirrored volume that is disabled because of a trivial, transient error, such as a disk that was switched off or otherwise unreachable when the volume was started. Most of the contents of the disk are probably identical to the contents of the other plex(es), but after the disk is back online VxVM will nevertheless have to resynchronize the whole volume because it simply did not keep track of which regions is changed in the remaining active plexes.

The same problem turns up when a plex was detached from a volume on purpose, in order to create a snapshot from the volume. Such so-called point-in-time copies, or PITCs, are often used for creating consistent backups that represent a specific point in time rather than the usually unrecoverable mess that a backup made from an active file system would create. Snapshots must be resynchronised before they are reused, and it would be nice if

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_7, © Springer-Verlag Berlin Heidelberg 2009

we could just resynchronize those portions of the volume that have actually changed (on the snapshot or the volume) since the snapshot was broken off.

7.1.1 What is a Log?

In order to resolve these and other problems the designers of VxVM came up with the clever idea of multi-column bitmaps that reside in separate, so-called LOG plexes. Each bit in the bitmap corresponds to a region (extent) of the volume and tracks one aspect of its state. For instance, whether it is currently undergoing write I/O or if it has undergone write I/O since a plex was disabled or detached. I.e. whether the contents of the plexes in the region are guaranteed to be synchronous or not relative to some other plex.

Logs are usually contained in a volume or they reside in a separate volume that is connected to its data volume by a pointer called a "Data Change Object", or **dco**. There are various logs for the diverse applications drafted above, and there is a quite modern log, the "**dco** version 20" that combines the most interesting features into a single data structure. We will discuss the creation of this very modern log in the "Easy Sailing" chapter. This log is easy to use and very powerful, but the **vxprint** output of a volume with such a log attached can be quite confusing. Apart from this, understanding the logged volume's behavior in case of special situations like crashes or mirror-splits is difficult without some serious work at the basis. Therefore, in the "The Full Battleship" we discuss the various other (older) types of logs, which for the most part are subsets of the **dco 20** log, so you understand what each log is meant for and what it does. In the "Technical Deep Dive" you will be shown what exactly happens inside VxVM when the logs are used and how they are used during recovery or resynchronisation. On the basis of this, you will be able to predict recovery and resynchronisation behavior for volumes containing or not containing a log of any given type. Some more information about logs can be found in the chapter on point-in-time copies (chapter 9) on page 238.

7.1.2 Simple Log Operations

The simplest way to prepare your volume for all possible havoc is the following command, issued on an existing volume (in this case our favorite, short-named **avol**):

```
# vxsnap prepare avol
```

Having done that, the volume is prepared for fast recovery after a system crash as well as for fast resynchronisation after a plex becomes disabled (due to I/O failure, for example), or detached (typically for a snapshot or point-in-time copy).

The difficult thing here is understanding the output of **vxprint**. Look at the volume before and after the command:

```
# vxassist make avol 1g layout=mirror init=active
# vxprint -qrtg 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
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c0t3d0   ENA

# vxsnap prepare avol
# vxprint -qrtg 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
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c0t3d0   ENA
```

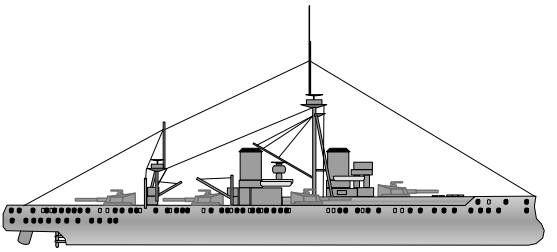
```
dc avol_dco     avol         avol_dcl
v  avol_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg03-01     avol_dcl-01  adg03    0        544      0         c0t4d0   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-01     avol_dcl-02  adg04    0        544      0         c0t10d0  ENA
```

Strange, isn't it? But when you parse the output top to bottom you will find just two volumes that stick together. So let's use **vxprint -rtL** to separate them for readability.

```
# vxprint -qrtLg 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
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c0t3d0   ENA
dc avol_dco     avol         avol_dcl

v  avol_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg03-01     avol_dcl-01  adg03    0        544      0         c0t4d0   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-01     avol_dcl-02  adg04    0        544      0         c0t10d0  ENA
```

Using this output format it becomes clear that the original volume has simply received a new type of object, the DCO (data change object called **avol_dco**), which was appended to it at the bottom. In addition, a new, redundant volume was created which is of minimal size. The small size results from the fact that only a bitmap is stored in the volume, and no data. That volume is the DCL (data change log), and the DCO is a pointer connected to the original volume that points to its associated DCL as can be verified by looking at the ASSOC column of the dc object: **avol_dco** is associated with **avol_dcl.**



The Full Battleship

7.2 Log Maintenance

Creating and Displaying a Volume with a Log

All logs are created equal! At least in so far as logs of any kind are created using a variant of the same **vxassist** command. You can add a log specification to any layout by appending "**,log**" to the layout parameter. For instance, to create a mirrored volume with a DRL (or dirty region log) you would replace the common syntax:

```
# vxassist make avol 1g layout=mirror init=active
```

with:

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

DRL is the default logtype for all volumes except RAID-5, where the only possible value is a **raid5log**. Displaying the new volume with vxprint will show that the volume has three plexes instead of two, which might lead to the assumption that the volume is now equipped with three mirrors. But closer examination will reveal that the third plex is actually not a data plex having the same size as the volume, but actually a plex which in place of the plex size is marked **LOGONLY**, and which only contains a very tiny subdisk. That subdisk holds a bitmap for initializing the dirty region map for **RDWRBACK**-synchronisation when a redundant volume is restarted after a system crash. Look at the output and find the log:

```
# vxprint -qrtg 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
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   LOGONLY  CONCAT    -        RW
sd adg05-01     avol-03      adg05    0        528      LOG       c0t11d0  ENA
```

Mirroring and Removing Logs

Just like data plexes, log plexes can be added and removed at any time during the lifetime of a volume. To add a log to an existing volume, use **vxassist** with the **addlog** subcommand:

```
# vxassist addlog avol # to insert a single additional log plex
# vxassist addlog avol nlog=2 # to insert two additional log plexes
```

This will add one or several DRL log plexes to any volume with a stripe or concat plex layout, and a **raid5log** to a volume with RAID-5 layout.

To remove a log plex from a volume use **vxassist** with the **remove log** subcommand, or just use the low-level command **vxplex -o rm dis <plexname>** (which can be used to remove any kind of plex - data or log).

```
# vxassist remove log avol # to remove a single log plex
# vxassist remove log avol nlog=2 # to remove all log plexes except two(!)
```

The latter command is somewhat confusing. When adding logs, **nlog=...** specifies the number of logs to add, but when removing logs, **nlog=...** does not specify the number of logs to remove, but the number of logs to leave attached to the volume. This is the same behavior as with adding and removing mirrors (which is actually obvious, since both are just plexes, albeit of different types).

Using Other than the Default Log Types

You may not want to use the default log type. The default is **raid5log** for RAID-5 volumes, DRL for all others. But those give you just logs that speed up recovery of a volume that crashed while active. To enable speedy resynchronisation of a volume with plexes that have been temporarily detached of disabled, e.g. because of loss of connectivity to a disk or storage array, or because a point-in-time copy was created and chopped off the volume, a different kind of log is required: a DCL (data change log). While a DCL is not just another plex in the volumes but actually a separate volume connected to the data volume by a special object called a DCO (data change object), it is still created by the same **vxassist** command syntax, just with an additional flag to specify the **logtype**. (BTW: The reason for the DCL residing in another volume has to do with more freedom in storage allocation for the log. There is no other magic reason about it.)

```
# vxassist make avol 1g layout=mirror,log logtype=dco # create volume with DCL
        # you can use "init=active" if you want, but we omitted it
        # (and may continue to do so) simply because of page space constraints.
```

Adding and Removing DCL Plexes to/from Existing Volumes

Adding and removing DCL/DCO logs to or from a volume is just as simple as adding and removing DRLs. But obviously we must specify the non-default log type of **dco** as the **logtype** using the **logtype=dco** parameter just as we do when creating a volume with a non-standard log. After all, how is VxVM supposed to know what kind of log we have in mind while we are typing the command? The following examples add and remove DCLs to and from a volume:

```
# vxassist addlog avol logtype=dco [nlog=X] # add X DCL(s) to a volume
# vxassist remove log avol logtype=dco [nlog=X] # remove all but X DCLs
# vxplex -o rm dis <plexname(s)> # removes the given plex(es) from DCL
```

The last command, the one that disassociated a plex (**vxplex dis**) and then removes it using the option **rm** (**-o rm**) is the most universal of all plex removal commands. It can remove any kind of plex: **log**, **data**, **dco**, sparse plex etc. in any state: **enabled**, **disabled**, **iofail**, **recover** etc. It is highly recommended to get used to the syntax for this command because it is so much more universal than the specific **vxassist** commands.

7.3 Details About Logs

7.3.1 DRL (Dirty Region Log)

Purpose of DRL

DRL logging serves exactly one purpose: to shorten the resynchronisation process of a mirrored volume that was open when VxVM's control over the volume was forcefully removed (e.g. by a system crash or by loss of access to the disk group due to I/O or path error). VxVM keeps a flag in the private region that tracks if the volume has received a write since it was opened. If that flag is set when a volume is started, then the implementation of VxVM (rightfully) demands the volume must be synchronised before use. This synchronisation is done via a **RDWRBACK** task as discussed in chapter 5, page 136.

Let's reiterate the process of **RDWRBACK** synchronisation shortly in draft form:

It begins with VxVM setting the access mode of the volume to **RDWRBACK.** The volume then becomes immediately accessible for reading as well as writing. In **RDWRBACK** access mode a write to a volume is done identically to the normal access mode. A read, however, is read from any of the volume's plexes according to its read policy (not read mode). The result of the read is not passed to the user right away, but instead it is first persisted (i.e. synchronously written) to all other plexes at the same offset. This makes sure the user will never get different data for the same block, thereby preserving the partition semantics.

When the volume is set to **RDWRBACK** access mode VxVM also allocates a bitmap representing the **CLEAN/DIRTY** state of every region of the volume (the region size that each bit represents may vary, but is roughly in the megabyte range). This bit is called the dirty region map (this is **not** the dirty region log!). Because it is unknown which regions actually are out of sync VxVM must assume the worst case, i.e. the case where every region is dirty, and will accordingly initialise all bits of the dirty region map to **DIRTY**.

When a region has been written to in the volume, the corresponding bit in the dirty region map is set to **CLEAN** because after the write the region is known to be in sync with the other plexes. Likewise, if a region has been read while in RDWRBACK mode it is known that the region must now be in sync, and therefore the region's bit is set to **CLEAN**.

As long as the volume is in RDWRBACK access mode the bitmap must be checked for every read or write operation, because it must be established if the I/O target region must be synchronised during the read and the bit reset, or if the write I/O must be followed by clearing the region's bit in the dirty region map.

Checking the bitmap for every I/O takes CPU time and induces latency, so it is desirable to eventually get rid of the dirty region map altogether. In order to do so, the volume must first be fully synchronised. For this purpose VxVM spawns a kernel thread for every volume

undergoing **RDWRBACK** synchronisation. This thread simply reads all the dirty regions of the volume and throws the results away. The (desired) side effect of doing so is that because the volume is in **RDWRBACK** access mode, its contents will be completely synchronised after the thread has finished. Therefore, the last thing the thread does is set the access mode to normal before it finishes.

This thread is what you see in the output of **vxtask list**.

The whole idea of a dirty region log is to be able to initialise not all the regions of the dirty region map with **DIRTY** flag bits, but just those that might actually **be** dirty. In that case, the kernel thread will be finished in a very short time because it will only synchronize a few hundred regions, which takes only a few seconds. This way, most of the time the resynchronisation will be completed upon boot even before the application actually starts using the volume, yielding in no performance impact whatsoever.

Implementation of DRL

DRL is implemented as a persistent bitmap (i.e. one that resides on disk), either in its own **LOGONLY** plex inside the volume, or as an additional column in a multi-column bitmap in a version 20 DRL. The theory for DRL access goes like this: Whenever a volume containing a DRL is written, then first of all the bit of the DRL that corresponds to the region must be set so that in case of a crash, the volume "remembers" that this region may be out of sync. This DRL write I/O must of course be synchronous, as we cannot allow the user data of one plex to reach the disk before the DRL I/O. This would leave a "write hole", where the plex is already out of sync, but the DRL does not yet reflect this. If the system crashed at that time, the partition semantics would be broken which is of course unacceptable.

In any case, only after the corresponding bit in the DRL (all plexes of it) is set, the write I/O is unblocked and allowed to continue. After completion of the write on all plexes, the bit can be reset to clean because the region is now once again synchronous.

If you looked at the DRL as a big panel full of lights - one light for each bit - the DRL would literally look like one of those "blinkenlights" panels from very old science fiction movies. The bits are constantly changing, but relatively few are on at any one time.

As is usual with a product such as VxVM there are a few things that are implemented differently from what the average programmer would do. One of these is the fact that there are actually two bitmaps in the DRL: one that is used during user I/O (the one discussed above). This is called the "active bitmap". And one that is used during recovery, which is called the "recovery bitmap". The latter contains only **CLEAN** bits during normal operation. When a recovery situation is encountered, VxVM combines the bits from the active bitmap with the bits in the recovery bitmap using the binary **OR** operation, i.e. it adds the bits from the active bitmap to the recovery bitmap. After that, the active bitmap is reset to **CLEAN** and recovery is started. In the unlikely event of another crash during the short period of recovery no information is lost, and no extra synchronisation overhead is encountered! VxVM will simply **OR** the new **DIRTY** bits into the recovery map and continue the (very short) synchronisation process.

Performance Implications of DRL

While common sense tells us that the write performance of a volume must suffer tremendously when a DRL is present, it is in fact not so. How can that be? The secret to this is the following:

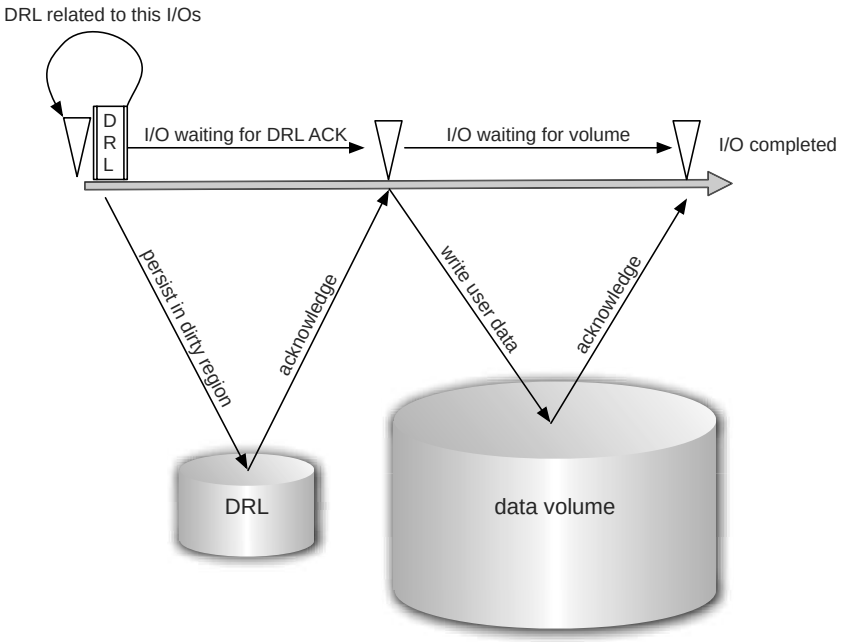


Figure 7-1: **A DRL introduces significant latency into volume I/O only on single, individual writes. A write I/O must first wait until its related I/O to the DRL has completed before it can start writing to the volume. In that case, performance is normally not an issue.**

While a single I/O must indeed first be prepared by persisting the corresponding **DIRTY** bit into the DRL, this overhead diminishes, even virtually disappears, when a long queue of I/O requests is processed. In that case, the first I/Os are analyzed and the corresponding bits in the DRL identified and set. The DRL is persisted and, while VxVM waits for completion of the DLR I/O, the next I/Os are already analyzed, their bits set in the DRL and the DRL I/O is sent to the HBA. As soon as the first DRL write completes the I/Os corresponding to that instance of the DRL are unblocked and sent to the HBA, too. While that user data is written, more write I/Os are accumulated in the queue, inspected, and the corresponding DRL bits are set. By the time the first user data I/Os come back completed, the second DRL write is returned from the controller so the second batch of user data I/Os unblocks. Also, the third DRL is sent to the controller, unblocking more user data I/Os.

In effect, DRL write I/Os are simply inserted into the (usually long) I/O queue so that in general no process has to wait for DRL write completion unless the machine does only

single, sparse write I/Os, in which case the performance is not usually limited by I/O anyway. Trying to make a graphical representation of this complicated matter has proven to be challenging. The picture we came up with does not look simple at all. But it is actually not much more than multiple instances of the former picture interleaved into one, as you will see upon closer inspection.

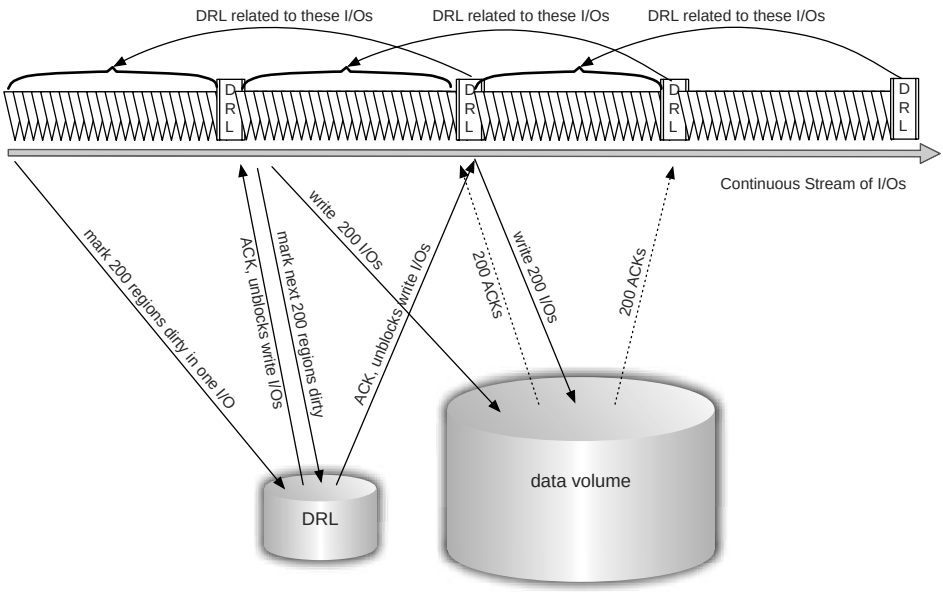


Figure 7-2: **When write I/Os are hammering on the LUNs DRLs are interspersed into the I/O queue. As soon as a DRL write (to all DRL plexes, of course) has been acknowledges from the storage the corresponding user data write I/Os are unblocked and can proceed. As those user data acknowledgements are collected from the HBA driver, the corresponding bits in the DRL become eligible to be reset. For performance reason, they are not reset immediately. Instead the number of "dirty" bits in the DRL is throttled at 200.**

One can estimate that DRL adds roughly about 5% of I/Os to the write I/O stream. This has to do with the fact that VxVM throttles the amount of **DIRTY** bits in a DRL to a maximum of 200. I.e. VxVM does not reset **DIRTY** bits right away as soon as they are eligible for it, but waits until the threshold of 200 **DIRTY** bits is reached, then cleans the ones whose I/Os have been returned as "completed" by the disk controllers on all plexes.

In other words, DRL operation inserts small write I/Os into the I/O stream each of which may unblock dozens to hundreds of user data write I/Os and introduce only minimal extra latency except for the pathological, but normally uncritical, case of an almost empty write I/O stream.

7.3.2 DCL/DCO (Data Change Log / Data Change Object)

Purpose of DCL

A DCL is meant for keeping track of changes that occur to a volume during the time that a plex is in a detached, disabled, offlined, or disassociated state, i.e. it does not normally take part in volume I/O when all plexes are **ENABLED** and **ACTIVE**. The usual reasons for such a removal from active participation in volume I/O are either a point-in-time copy or an I/O error of some sort.

In the case of an unrecoverable I/O error (e.g. a missing disk or permanent read errors) that leads to the disabling of a plex, a DCL is used to keep track of write I/O to the remaining plex(es) so that after the error situation has been cleared, the resynchronisation can be executed with minimal overhead. If a disk is permanently broken, then those regions of the disabled plex that resided on that disk must be synchronised along with all changes to other portions of the volume, where the original data was not lost and must simply be updated.

If a disk was only temporarily missing - it may have been disconnected or the paths to the disk may have gone bad - then after the error is fixed only the changes need to applied to the plex during resynchronisation. Since the original data was not lost, this suffices to bring the plex contents up to date with the volume contents.

In the case of a point-in-time copy the plex is removed from the volume and another volume is "wrapped around" the plex. This new volume, the so-called snapshot-volume, can then be mounted and its contents can be backed up or serve some other purpose. In this case we have to deal with two different volumes, each of which can be written to. It is therefore obvious that the disassociation of a plex from its volume for snapshot purposes requires a DCL plex to be disassociated along with the data plex. If this is satisfied, then VxVM will keep track of changes not only in the original volume but also in the snapshot volume. Remember the snapshot volume can be used read-write.

Resynchronisation speed of snapshots is important because such point-in-time copies are often reused at regular intervals, typically for daily backups, and so their contents must be resynchronised frequently. Resynchronizing just the differences to the previous contents usually saves a great deal of time and I/Os. As a rule of thumb, it is unusual for a database to change more than five percent of its contents er day. Accordingly, resynchronisation would take twenty times longer without DRL than it would with DCL.

Because the creation of the correct amount, type and location of DCLs for all the purposes listed above and the correct modifications to the VxVM objects when snapshotting a volume are not trivial there is a new command that will create DCLs for you in such a way that they automatically do the The Right Thing™. And because the most complicated case is the case of the snapshot volume, the command is called **vxsnap**. Here's what you do to create a DCL that just works:

```
# vxsnap -g <DG> prepare <Volume>
```

(We are actually cheating a little here, but for you benefit. The command we just used, **vxsnap prepare**, will not create a pure DCL, but a DCL with a DRL included in a so-called

multi-column bitmap. For the sake of the argument, we'll just pretend for the rest of the chapter that **vxsnap prepare** simply added a standard DCL. Creating standard DCL logs is done via **vxassist addlog logtype=dco**, but there is a lot of extra stuff you would have to think about in order to make it work right with snapshots, so let's just stick with the syntax we just used.)

We looked at the objects created by this command in the Easy Sailing section before, but let's parse it more thoroughly this time. We'll start with a new, mirrored volume **avol, and we'll use the -L flag in addition to the normal flags to vxprint** in order to separate the volumes so they are easier to read.

```
# vxassist -g adg make avol 1g nmirror=3 init=active
# vxprint -qrtLg 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
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

# vxsnap prepare avol
# vxprint -qrtLg 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
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
dc avol_dco     avol         avol_dcl

v  avol_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-01     avol_dcl-01  adg04    0        544      0         c0t10d0  ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg05-01     avol_dcl-02  adg05    0        544      0         c0t11d0  ENA
```

As you can see, an additional volume was created, with as many plexes as the original volume. This new volume is the DCL volume, which is obvious from its size (very small, just big enough for a bitmap), and its name, **avol_dcl**. But another object has been created as well: if you look at the end of the original volume, there is now a new, as of yet unknown object called **avol_dco**. Its type (first column) is given as **dc**, and the name stands for "**avol's data change object**". This object points to the DCL and thus connects the two volumes.

To jump ahead to the chapter about snapshots, let's look at what happens when we actually add another plex in preparation for a snapshot action:

```
# vxsnap addmir avol
# vxprint -qrtLg 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
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
pl avol-04      avol         ENABLED  SNAPDONE 2097152  CONCAT    -        WO
sd adg06-01     avol-04      adg06    0        2097152  0         c0t12d0  ENA
dc avol_dco     avol         avol_dcl

v  avol_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-01     avol_dcl-01  adg04    0        544      0         c0t10d0  ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg05-01     avol_dcl-02  adg05    0        544      0         c0t11d0  ENA
pl avol_dcl-03  avol_dcl     DISABLED DCOSNP   544      CONCAT    -        RW
sd adg06-02     avol_dcl-03  adg06    2097152  544      0         c0t12d0  ENA
```

Another plex has been added to avol, and another plex has been added to avol_dcl as well. This plex is disabled because it currently has no meaning. The extra data plex is not used for reading anyway. To understand why this is the case, check out its state in the right most column: **WO** means Write-Only. This plex will only become **RW** (Read-Write) when it is put into its own, snapshot volume. As long as it is just a part of a data volume, ut not an actual mirror, it is kept in **WO** state. That means there is no way VxVM would write to just the snapshot plex but not the mirror, and so the DCL need not be active. Sounds difficult? Well, it is, but you can trust us on this.

Now let's look at what happens when a snapshot is actually created using a command with a very arcane-looking syntax (this syntax is actually necessary in order to create snapshots of multiple volumes at exactly the same point in time, but it sure does look ugly):

```
# vxsnap make source=avol/new=SNAP-avol/plex=avol-04
# vxprint -qrtLg adg
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  ROUND     -        fsgen
pl avol-04      SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg06-01     avol-04      adg06    0        2097152  0         c0t12d0  ENA
dc SNAP-avol_dco SNAP-avol   SNAP-avol_dcl

v  SNAP-avol_dcl -           ENABLED  ACTIVE   544      ROUND     -        gen
pl avol_dcl-03  SNAP-avol_dcl ENABLED ACTIVE   544      CONCAT    -        RW
sd adg06-02     avol_dcl-03  adg06    2097152  544      0         c0t12d0  ENA
sp avol_snp     SNAP-avol    SNAP-avol_dco
```

```
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
dc avol_dco    avol         avol_dcl

v  avol_dcl    -            ENABLED  ACTIVE   544      SELECT    -        gen
pl avol_dcl-01 avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-01    avol_dcl-01  adg04    0        544      0         c0t10d0  ENA
pl avol_dcl-02 avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg05-01    avol_dcl-02  adg05    0        544      0         c0t11d0  ENA
sp SNAP-avol_snp avol       avol_dco
```

The extra DCL plex that had been created by the **vxsnap addmir** command has moved - along with the extra plex created by the same command - into the new volume with the name SNAP-avol. There it serves the purpose of keeping track of changes that VxVM does to the snapshot volume. It does so on any machine - you can use this snapshot on a different host. Check the chapter about snapshots to learn more about how this is done.

In addition to the movement of the data plex and DCL plex into the newly created SNAP-avol volume VxVM also created an "**sp**" object for each of the two volumes concerned, and appended that object to the volumes. An **sp** object is a "snapshot pointer". It connects the snapshot children to their snapshot parents, so that VxVM can find the way home to the original volume when you want to resynchronize a snapshot with its parent.

Implementation of DCL

The DCL has been implemented as a bitmap with user-definable region-size, i.e. the size of the volume region each bit stands for can be adjusted according to the expected behavior of the application that uses the volume. If you used a very large region size, then you would create only few I/Os to the DCL but on the other hand, the log would fill up rather quickly with **DIRTY** flags if a lot of random writes occur. This would not make resynchronisation very much faster. If you used a very small region size, then most write I/Os would also create a write to the DCL if it is active (i.e. a plex has been deactivated from participating in the volume traffic). This would have an adverse affect on write performance, while on the other hand optimize resynchronisation to its maximum.

You can find much more detailed information about DCL/DCO in the chapter about point-in-time copies beginning on page 233.

Performance Implications of DCL

After the previous paragraphs it should be obvious that there are no performance implications for a volume that has a DCL attached during normal operation. Only if a plex has been split off from the volume or somehow disabled or stopped will a DCL be written to. In these cases the minimal extra I/O incurred by writing the DCL's bitmap is more than made up for by the I/O saved during the following resynchronisation cycle. One should therefore not hesitate to add DCLs to mirrored volumes unless one is afraid of the extra complexity in parsing **vxprint** output, or in the case that single plex deactivation is extremely unlikely or the volumes are so small that resynchronisation is not an issue (as a rule of thumb, resynchronisation speed, because it is throttled by VxVM, is usually around one or a few GB per minute, and the amount of throttling can only be increased, not reduced, by the user).

7.3.3 raid5log

Purpose of a Raid5log

Although we like to remind you that doing RAID-5 in software is not generally a good idea, for the sake of completion we will take a short look at the log for this type of volume. Why is a log for RAID-5 volumes necessary? In order to find out why a RAID-5 log is absolutely critical to have (and therefore created by default when you make a RAID-5 volume using the **vxassist** command) as well as what it should contain it is sufficient to look at a simple worst-case scenario: When a disk failure during a full-stripe write to a RAID-5 volume leads to a system panic!

When that happens, and the system comes back up and starts VxVM, then VxVM finds that one of the RAID-5 volume's columns is missing due to the faulted disks. It will allow user I/O on the volume, because the volume's RAID-5 plex has built-in redundancy.

But that is not enough!

Because there is a possibility that a write had been interrupted by the crash the parity data is not really reliable. Why is that? Consider a RAID-5 stripe consisting of 5 columns (including parity). A write is done that covers the whole stripe, but only columns zero and one are actually written, while a funky response from the failing disk for column two causes the system to crash and thus columns two and three as well as the parity column are not written because the system crashes before it can commit the writes. The volume is now in an unreliable state, because when data is read from the failed column VxVM must reconstruct the data by – remember? – reading all the remaining columns plus the parity column, and then XORing the data to extract what was originally stored on the failed column.

Unfortunately the reconstruction is done using mixed old and new data: some columns have already been updated, some have not, and the parity data does not correspond to this mixed state: the old parity data is valid only in combination with the old data; the new parity data is valid only in combination with the new data, but neither is valid with a mixture of old and new data! The whole calculation yields nonsensical data as a result. This nonsense will likely cause the operating system to panic, which is of course not desirable but you deserve it if you use software RAID-5 (we told you so ;=).

In addition to the very undesirable behavior of possibly causing an operating system panic, VxVM must also recalculate all the parity data once the failed disk has been replaced, because old and new data could have been mixed at any place; there is no record of it.

So to protect you from this unattractive scenario (if you still want to use RAID-5, that is), use the default VxVM behavior and let VxVM create a raid5log for you when you make a RAID-5 volume.

Implementation of Raid5log?

A raid5log must reside on a disk that is not part of the RAID-5 plex (otherwise the problem would still be there if **that** disk were to fail). It is organized as a circular buffer that contains enough space for several full stripe writes (five of them the last time we looked). It works much like a file system intent log: Any write to the volume is done by first writing the data to the raid5log. Behind the data some information is written that marks the data as valid (because we do not want to replay interrupted transactions onto the volume). After the write to the raid5log has been completed, the written data is considered safe and will (eventually) be persisted to the actual RAID-5 volume. There is no big hurry to do so unless there are many writes hitting the volume and the log would overflow.

If the system crashes (with or without a disk fault), then VxVM will replay the raid5log data to the volume upon volume start. The raid5log contains enough data to identify which data goes where, which data is valid, and the order in which the individual writes must be replayed. So using a raid5log we end up with two advantages:

1) there is no need to recalculate all of the parity information

2) the volume will actually work – because it contains valid data – and not cause the operating system to panic over and over again.