CHAPTER 9: POINT IN TIME COPIES (SNAPSHOTS)

by Dr. Albrecht Scriba

9.1 Overview

Several tasks, like backups or lengthy data analysis jobs, require a frozen image of the application data while the application remains online and produces new data. For instance, reporting needs a stable state of information to generate coherent and usable data. A data copy must be sent to a testing data center with a specified time stamp. Or, as another example, you could wish to create backup images that require a reduced amount of recovery complexity after a restore operation (see notes below).

9.1.1 Types of Snapshots

Veritas Storage Foundation provides several techniques to create a "frozen image" copy (also called “snapshot” or "point in time copy") of the current data set with different concepts, advantages, and disadvantages. Some techniques create the snapshot on the raw device, i.e. the volume layer, working with all data structures stored on the device: ufs, vxfs or other file systems, mounted or unmounted, tablespaces on the raw device itself, and so on. These can be referred to as volume-based snapshots.

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_9, © Springer-Verlag Berlin Heidelberg 2009

On the other side, the Veritas File System contains two mechanisms to create a file system based snapshot. These are file system-based snapshots, and the correct term in the Veritas world is "storage checkpoint".

Another way to classify snapshot procedures is by the expression pair physical – logical. A **physical** snapshot not only looks like a complete snapshot of all data, it is actually a complete copy of the data set. The advantage of a physical snapshot is its capability to be exported to another host without losing its snapshot function. I/O to the snapshot accesses only the snapshot device, the still running application based on the original device does not suffer performance degradation from snapshot I/O. On the other hand, as a disadvantage, the physical snapshot requires storage for a complete copy, and it takes remarkable time to synchronize its data from the original when created or refreshed.

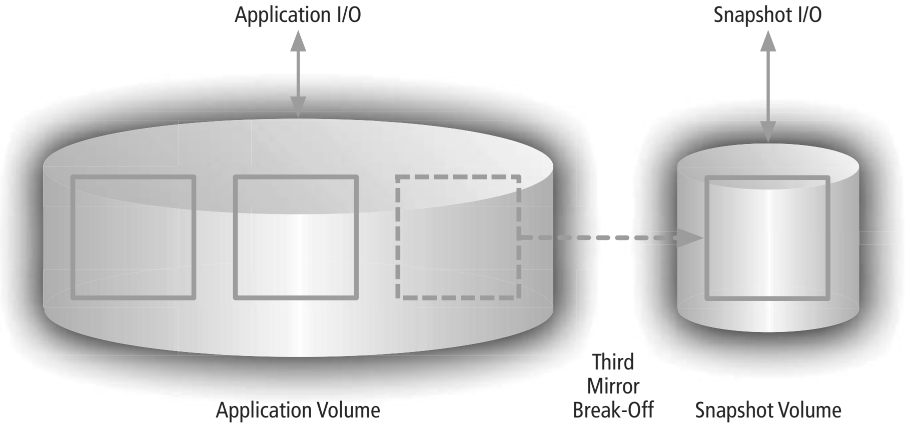


Figure 9-1: Physical raw device volume snapshot ("Third Mirror Break-Off")

A **logical** snapshot must simulate a complete snapshot, it only looks like, but it is not a complete frozen data copy. The underlying technique always combines two ways of accessing data through the snapshot. Data still unchanged since the creation of the snapshot are read from the original device, the snapshot device only points to the corresponding regions of the original device. In other words: An unchanged data set physically exists only once, but is accessed by both the original device and the snapshot. If the application wants to modify its data, the logical snapshot needs to store the original version of the data to be modified, before the new data set can be written to the application device (this is called "copy on first write", sometimes “copy before write” or “copy on write”). Apparently, the logical snapshot needs physical storage as well, but only in an amount sufficient to store the originals of application data modified since the creation and until the planned destruction of the snapshot. A logical snapshot therefore serves much better for temporarily limited tasks, such as backups or data transfers to another location. Furthermore, the concept of pointers to the original data set when creating a logical snapshot ensures, that the snapshot is available immediately. Note as a disadvantage of a logical snapshot its ongoing binding to the original device, snapshot I/Os are in many cases I/Os actually on the original device degrading application performance, and its physical export to a different host for offhost processing is impossible.

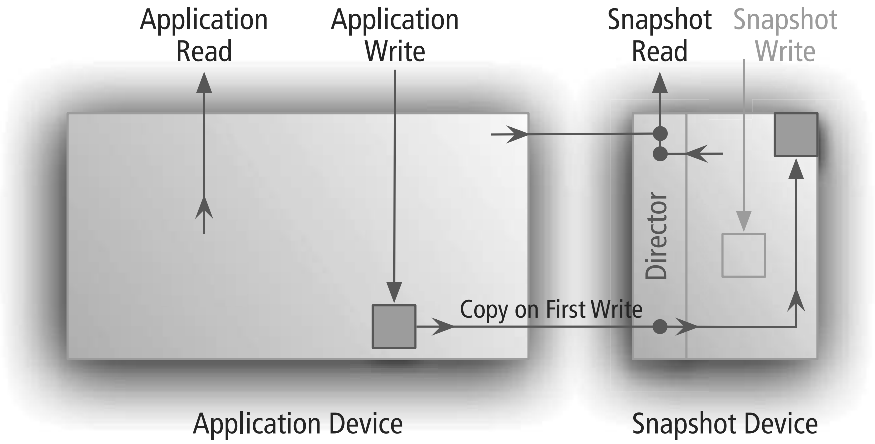


Figure 9-2: Logical Snapshot

9.1.2 Consistency Problems for Snapshots

If as the snapshot is taken while the application is running on the original device or file system, the snapshot does not provide a consistent copy from the application's point of view. This is because the snapshot, if it is taken from a live application's file system, is not fully consistent on its own, but only in combination with the application's internal state and the file system's buffer cache. But neither application state nor buffer cache become part of the snapshot. So we would need to either quiesce the application and unmount the file system, which is not usually desired), or use some magic to fix some or all of the inconsistencies of the snapshot later.

Two examples will illustrate the point:

If an application is based on a mounted file system, the file system state flag in the main super block is "active", signifying that the file system is not "clean" due to data blocks in the file system cache that have not yet been flushed to persistent storage. If the file system is not cleanly unmounted (e.g. the system crashes), the "active" state forces a file system check, otherwise the **mount** system call will refuse to mount the file system. A raw device snapshot taken from a mounted file system contains a raw copy of all device data, so its main super block carries the "active" flag. A file system based snapshot, on the other side, could perform an (automated) file system check on the snapshot data set after having created it, because this technique is aware of the existence of the file system. That is the "magic" we talked about above. But nevertheless, even with a checked snapshot file system, you only get a "clean" file system data set, not necessarily a clean application data set, because application write I/Os storing application transactions could consist of several file system I/Os to different files. A file system snapshot does not know about the application's internal logic, so it cannot provide any "magic" here. Instead, we must rely on the application's ability to recover from a crash.

What does a snapshot have to do with a crash, you may ask? Well, a snapshot of a volume or file system contains all of the persistent (i.e. the information that has been written to disk) information about a file system, but none of the transient information (i.e. residing only in memory). It is in this way is identical to the contents of a file system or volume that has crashed. (The difference being that even if a snapshot volume contains of several plexes, these plexes will not need to be resynchronized, which they do in case of a crash.) Most enterprise applications do provide for safe crash recovery, and for these, using snapshots should not be an issue. However, you must still be aware that recovery procedures must be applied when using a snapshot that was taken from a live file system.

A database using a raw device as storage without an intermediate file system layer optimizes performance by caching data in a sometimes large memory cache. Those data are flushed ("checkpointed") to the raw device from time to time (asynchronous I/O). To avoid loss of new data created by the last write transactions before a system crash, the database writes transactions which modify data nearly synchronously in a symbolic manner to a log device (called the **redo-log**). A database software that fulfills enterprise needs even in case of a crash must replay all synchronously stored transactions to the asynchronously flushed database structure starting from the point of the last database checkpoint.

Now assume a snapshot taken from an online database: the database structure and the transaction log do not carry the same timestamp, indicating that recovery is needed before opening the database. Here, too, the database must apply its (crash) recovery procedure to roll the redo log forward, thus integrating the most recent changes into the database.

To sum up: As long as the snapshot mechanism does not provide full application awareness including application recovery strategies, it cannot create a consistent snapshot of the data set of a running application. To actually get a consistent point in time copy, you must cleanly stop the application and, if based on a file system, unmount it before taking the snapshot. This limitation is valid under all circumstances: whether using software or hardware volume management, because they both suffer from the split between transient information in the kernel and persistent information on disk.

One way to overcome this limitation is to integrate the kernel buffer cache and application memory into the snapshot layer. This can only be done in a virtualized environment, in which the snapshot software can – at least theoretically – cooperate with the virtualization software to flush the relevant memory pages into the snapshot when it is taken and therefore maintain a higher level of data integrity. But storage management for virtualized hosts is only evolving now, and there is not much experience available yet.

9.2 Physical Raw Device Snapshots

9.2.1 Overview

A physical snapshot requires an extra copy of the volume data or, in terms of VxVM objects, an extra synchronized plex within the volume. Like all complete plex synchronization processes, this means a lot of I/O with system and application performance drawbacks and a certain amount of time (current hardware does around 1 GB per minute). Repeating that for every backup every day sounds rather wasteful, and it is.

In order to overcome both the complexities of creating new mirrors and separating them from their originals, then creating new volume objects

By the time VxVM 4.0 was being developed many new snapshot types and features had been developed and required elegant integration into the VxVM command structure. One of the most important older snapshot features (introduced in VxVM 3.2), the DCO (“data change object”) with its data change log volume to dramatically improve snapback performance (explanation will follow), was made the default for all volume-based snapshots. Therefore, creation of another volume data copy for snapshot purposes should be prepared with an associated DCO log volume to get the full snapshot feature set. This is done using the **vxsnap prepare** command:

```
# vxsnap -g adg prepare avol [alloc=<disklist>]
```

We have now added a DCO log volume to our data volume. If we specified the alloc parameter with a list of storage objects (disks, controllers, enclosures, etc.), VxVM will have used only those storage objects to place the new DCO log volume's subdisks on.

In addition, VxVM has set some important internal variables to the appropriate values (e.g. the "fastresync" flag was set to "on"). But a new plex, a new instance of the data, has not been created yet. To create it, we issue another simple command, the **vxsnap addmir** command. This will create, and start synchronisation of, another data plex that can later be separated from the data volume to live its own life as a snapshot volume:

```
# vxsnap -g adg addmir avol [alloc=<disklist>]
```

Again, we can specify certain storage objects to place the new subdisks on. Only this time, because the data plex is allocated the storage allocation controls where the data plex's subdisks are created rather than the DCO log volume's subdisks.

OK, so now we have a data volume that is prepared for snapshotting by adding a DCO log Volume and another data plex. Now we can simply turn the data plex into a separate snapshot volume by "snapping it off" the data volume. This is again just one command, (albeit with a weird looking parameter, as you will see). To snap a plex off into a snapshot volume use the **vxsnap make** command. Here is an example:

```
# vxsnap make source=avol/new=SNAP-avol/plex=avol-03
```

This creates a volume which is separate from the source volume (**source=avol**), gives it the new name SNAP-avol (**new=SNAP-avol**) using the data plex avol-03 (**plex=avol-03**). You can now use that new volume, the SNAP-avol. It contains an exact copy of the data volume at the very moment the **vxsnap make** command was run. Be aware that file system and application data recovery is required which is equivalent to the recovery after a system crash (see introduction).

At any time you can refresh the contents of the snapshot volume using the **vxsnap refresh** command. The most common use for refreshing is to update a snapshot just before it is backed up. Here's an example for refreshing:

```
# vxsnap refresh SNAP-avol source=avol
```

All data blocks that have been changed in either the snapshot volume **SNAP-avol** or the original volume **avol** will be read from **avol** and copied into the appropriate regions in **SNAP-avol** by running the **vxsnap refresh** command.

Because data is copied to the target **SNAP-avol** at block level (i.e. into the raw device), it cannot be done while SNAP-avol is mounted, of course. Your file system device driver will say "thank you for not totally confusing me".

9.2.2 A Look at What Goes on Inside

In order to understand snapshots we need to reiterate what happens when we add another data plex to a mirrored volume. We assume you know what a mirrored volume looks like in the **vxprint** output, and start with the added mirror. Here's what you get:

```
# vxassist -g adg [-b] mirror avol [layout=<layout>] [<storage-attributes>]
# vxprint -rtg adg avol
[…]
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         c1t1d0   ENA
```

```
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1  ENA
pl avol-03      avol         ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg03-01     avol-03      adg03    0        2097152  0         c1t1d2  ENA
```

Pretty simple and pretty obvious: a third plex was added, along with its subdisk, and that's it. Theoretically, we could now use the appropriate low-level commands for creating an empty volume object, and for disassociating the third plex from the original volume and attaching it into the newly created one. We would thus obtain a new volume initialized with the data contents of the original volume at the time that we disassociated the third plex. But doing so requires a lot of know-how about creating and handling low-level objects. So a long time ago Veritas created an easy to use front-end for creating snappable plexes. We could actually use this now deprecated form of snapshot commands which are subcommands to **vxassist**. For completeness, this legacy version will be covered in its own section later in this chapter. But because its interface and objects were developed over a long time the concepts are less easy to grasp than they are with the new approach which uses the new **vxsnap** command. So let us now jump way ahead in the development of VxVM and right into the most advanced snapshot mechanism in Volume Manager.

Let's first look at what happens when we prepare a volume for snapshotting:

```
# vxsnap prepare avol
# vxprint -rLtg adg avol
[…]
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         c1t1d0  ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1  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         c1t1d2  ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -       RW
sd adg01-02     avol_dcl-02  adg01    2097152  544      0         c1t1d0  ENA
```

This command added some new VxVM objects with funny names. In particular, a tiny volume was created, with the name **avol_dcl**. The name DCL stands for "Data Change Log". It is a log that keeps track of changes to a volume. However it does not store the actual data but just sets the appropriate bit in a multi-column bitmap corresponding to the region in the data volume that incurred a change. Because the volume needs to update the DCL bitmap when it writes, the volume object must contain information pointing to the DCL volume. This pointer is the "**dc**" object that was added to the volume (last line of the avol output).

This sounds rather confusing so let's draw the output into an image that is probably easier to understand. We will skip the plex internal structures, i.e. the subdisks. Their grouping within the plex is irrelevant for nearly all snapshot features.

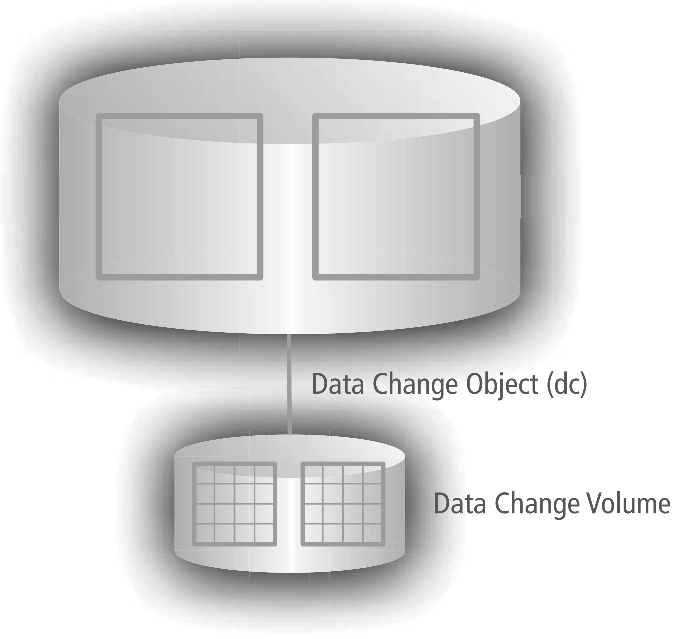


Figure 9-3: Snapshot prepared application volume with Data Change Log (DCL) volume linked by a Data Change Object (DCO)

OK, again: the application data volume (top) is linked to a very small volume (bottom) with two plexes. You will not find a device driver for the small volume, it only serves for VxVM internal purposes and does not contain any application data. Actually it contains a mirrored multi-column bitmap, which among other things logs regions of the top-level volume affected by write I/Os. Because each bit position in the multi-column bitmap corresponds to a large region in the data volume the plex is drawn as a grid. We will explain further details of the DCO in the "Full Battleship" and the "Technical Deep Dive" part.

We still need to add another plex to get a volume data instance for the snapshot. The command **vxsnap** provides a keyword to add a mirror to both the data volume (top) and DC log volume (bottom). We used that in the introduction of this chapter and will now look at what objects are created by it:

```
# vxsnap -g adg addmir avol [alloc=<disklist>]
# vxprint -rLtg adg avol
[…]
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         c1t1d0  ENA
```

```
pl avol-02      avol         ENABLED  ACTIVE   2097152 CONCAT   -       RW
sd adg02-01     avol-02      adg02    0        2097152 0        c1t1d1  ENA
pl avol-03      avol         ENABLED  SNAPDONE 2097152 CONCAT   -       WO
sd adg03-02     avol-03      adg03    544      2097152 0        c1t1d2  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        c1t1d2  ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544     CONCAT   -       RW
sd adg01-02     avol_dcl-02  adg01    2097152  544     0        c1t1d0  ENA
pl avol_dcl-03  avol_dcl     DISABLED DCOSNP   544     CONCAT   -       RW
sd adg02-02     avol_dcl-03  adg02    2097152  544     0        c1t1d1  ENA
```

Application and DC log volume simply acquired another plex. Furthermore, note the difference in the application state (SNAPDONE) and in the access mode of the new top-level volume plex (WO = write-only) compared to the standard **`vxassist mirror`** command above. SNAPDONE only means, that the plex is marked for snapshot, its write-only access does not modify the regular read policy of the volume. The corresponding DC log volume plex is in DISABLED kernel state (no I/O possible, explanation see below in the latter sections of this chapter) and DCOSNP state, which marks the plex in the same manner as the SNAPDONE state of the top-level volume plex for snapshot purposes.

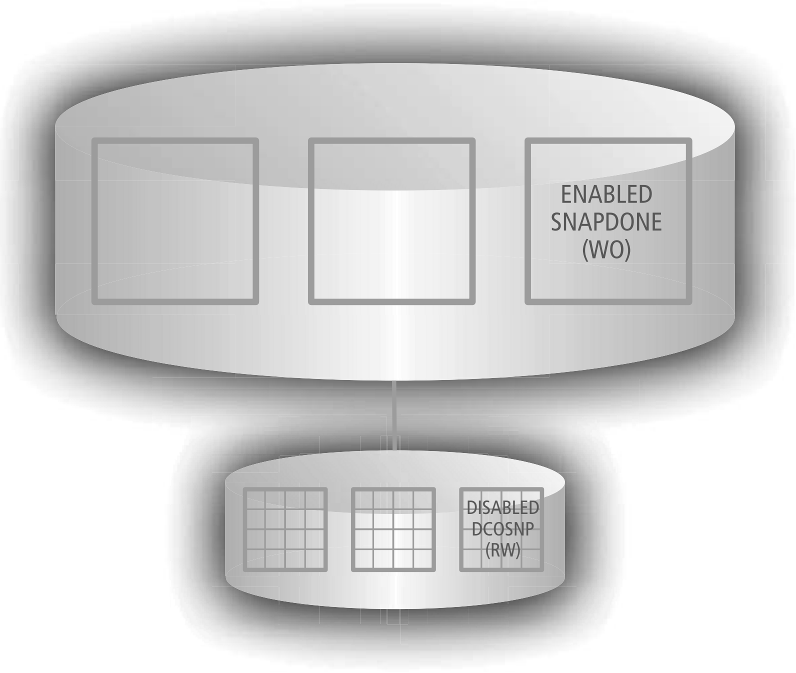


Figure 9-4: **Snapshot Prepared Application Volume with Data Change Log Volume and Third Mirror**

Currently the snapshot plex is still a full member of the volume except for read access from the plex being prohibited – the snapshot plex remains WO, or write-only. But its data changes synchronously with the other plexes. In other words: This plex is still live, it is not yet a snapshot but is only prepared to become a snapshot. To actually create the snapshot, to split it from the data volume, we need to enter a somewhat weird-looking command (we will explain the strange slash-separated parameter syntax later):

```
# vxsnap -g adg make source=avol/new=SNAP-avol/plex=avol-03
# vxprint -rLtg adg
[…]
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  ROUND    -        fsgen
pl avol-03      SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT   -        RW
sd adg03-02     avol-03      adg03    544      2097152  0        c1t1d2   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 adg02-02     avol_dcl-03  adg02    2097152  544      0        c1t1d1   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         c1t1d0  ENA
pl avol-02       avol          ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg02-01      avol-02       adg02    0        2097152  0         c1t1d1  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         c1t1d2  ENA
pl avol_dcl-02   avol_dcl      ENABLED  ACTIVE   544      CONCAT    -       RW
sd adg01-02      avol_dcl-02   adg01    2097152  544      0         c1t1d0  ENA
sp SNAP-avol_snp avol          avol_dco
```

We already know that an easy understanding of snapshots at a glance is quite difficult. But once again, drawing an image based on the disturbing ASCII command output does help indeed. What happened when we split the snapshot from the data volume was this:

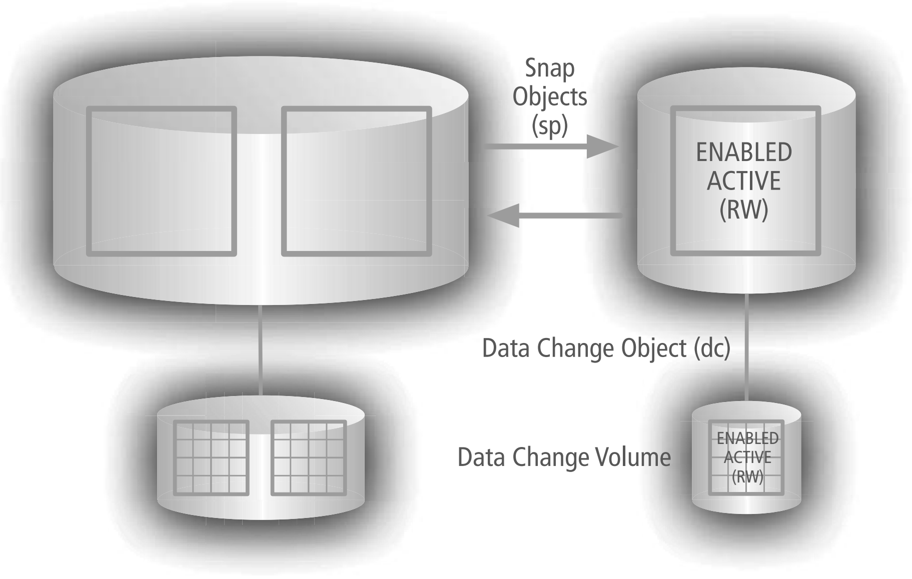


Figure 9-5: Application Volume with Split Snapshot Volume

Both plexes, the data plex (top) and the DC log volume plex formerly marked as SNAPDONE and DCOSNP respectively, have been broken off from their original volumes. They both changed their state to ACTIVE. Additionally the data plex changed to read-write access mode (RW) and the DC log plex changed its kernel state to ENABLED. Each was wrapped into a volume object. This was done to add application access to the snapshot volume (there are now device drivers for **SNAP-avol** in the **/dev/vx/[r]dsk/adg** directory) or to form a DC log volume. The DC log volume is linked to the snapshot data volume by a new DC object called **SNAP-avol_dco** (<**snapvol>_dco** in general) in the same manner we already mentioned for the running application volume. And finally, both data volumes are cross-linked by snap objects (type "sp" in the first column of the **vxprint** output) to enable the snapback procedure: the snapshot volume together with its associated DC log volume turns back into a synchronized member of the running application volume for further snapshot tasks (see the "Full Battleship" section below).

Note that without specifying storage attributes the snapshot volume and its associated DC log use subdisks placed on two different disks (here **adg02** and **adg03**) and that the remaining original volume uses some of these disks as well (**adg02** and **adg03**). This makes offhost snapshots impossible, as will be explained in the "Full Battleship" section.

The snapshot volume can be used just like any standard application volume: it can be mounted, accessed by another instance of the application, and so on. Its associated DC log volume ensures that write access to the snap volume is tracked and considered when snapping it back to the original volume or when refreshing the snapshot (any region that was modified in either the snapshot or the data volume needs be resynchronized). But snapping back and refreshing snapshots will be covered in the "Full Battleship" section.

To conclude with the main features of the volume based raw device snapshot mechanism:

1. The snapshot copy is physically spoken completely independent from the original device, thus snapshot I/O does not degrade application performance, and offhost processing is possible.
2. The snapshot can be accessed in read-write mode (will be explained later).
3. The snapshot can be used for immediate recovery of both, corrupted application data and physically damaged devices (explained later).
4. The snapshot technique is independent from the data structure (e.g. file system) on the device.
5. The snapshot function is protected against application and system crashes or disk group deports and imports (details worked out later).
6. The snapshot requires the storage for a complete copy of the original device.
7. Data must be completely synchronized or resynchronized before the snapshot can be created. This can take a business critical amount of time.

vxsnap's Weird Syntax

We promised earlier that we would explain the funny syntax on the **vxsnap** command line. Remember we had to cope with commands like this:

```
# vxsnap make source=avol/new=SNAP-avol/plex=avol-03
```

You may have thought "what were the developers smoking?" (or even: "I want some of the same stuff"!). But that would not be fair. They were actually being very smart people.

You see, creating a snapshot of a volume is easy, and would not require such funny slash-separated 3-tuples. However, what if you need to create snapshots of a great number of volumes, and they all need to be consistent with each other? You cannot rely on many snapshot commands simply executing in rapid succession and hope that there will be no inconsistencies. That would be totally inacceptable from an enterprise perspective.

Veritas' solution to this problem is to make the vxsnap command accept multiple snapshot volume source and destination tuples (triples in this case). Then, when the command is executed, they all snap at precisely the same point in time. In this case, consistency is guaranteed rather than approximated. While the cost of this approach is merely an atypical parameter format, its benefit is immeasurable.

9.2.3 A Logical File System Snapshot

Another snapshot technique with a mostly inverted set of features compared to the above mentioned procedure is worth being explained in the Easy Sailing section: the legacy VxFS snapshot. It belongs to an older concept of snapshots and is not generally used in Veritas installations any more because VxFS offers vastly superior approaches today. However, most other common snapshots use a very similar concept. E.g. Solaris UFS snapshots or MS-Windows file system snapshots work on the same basis as the legacy snapshot file systems discussed here. They all share the huge drawback that they are not crash-proof, i.e. if the system holding the snapshot incurs a fault and crashes, the snapshot is lost and a new snapshot must be taken. While this does not sound too bad, keep in mind that this also means there is no way of ever getting the exact state of the file system back that we had at the time the snapshot had been initialized. This may well be a show-stopper for an enterprise evaluating snapshot mechanisms!

The technique is not only bound to the file system driver code, it is also a so-called logical snapshot, that is, unchanged data remains stored on the original device and is accessible by both the original device driver and the snapshot driver. Data that has been written, however, is first copied to the snapshot and subsequently overwritten on the original device. The snapshot device itself does not contain a complete file system, but just references: to the original data for all unchanged regions and to its own data store for the blocks that have been saved from the original before they were overwritten.

The physical snapshot device must provide storage capacity only for the originals of modified data (10% per day are sufficient in most cases).

As a data store for the snapshot file system you can use any device appropriate to serve as a base for VxFS (such as logical volumes of other software manufacturers, partitions, USB sticks, even RAM disks). Nevertheless, for our convenience in a Storage Foundation book, we choose Veritas volumes in the following explanation and demonstration.

First, we create the original device and file system, mount the latter and place a scratch file on it:

```
# vxassist make avol 1g layout=mirror init=active
# mkfs -F vxfs /dev/vx/rdsk/adg/avol
    version 7 layout
    2097152 sectors, 1048576 blocks of size 1024, log size 16384 blocks
    largefiles supported
# mount -F vxfs /dev/vx/dsk/adg/avol /mnt
# mkfile 10m /mnt/file0.10m
```

In order to create a VxFS snapshot, we need a considerably smaller cache device (we choose 10% of the original device, less than 5% are not supported). By mounting it with the special option **-o snapof=<original-blockdevice>|<original-mountpoint>**, we are telling the VxFS device driver to initialize the appropriate data structures and establish the snapshot; we do not need to place a VxFS on it before.

```
# vxassist make cacheavol 100m layout=mirror init=active
# mkdir /mnt_snap
```

Create the snapshot by using the original block device:

```
# mount -F vxfs -o snapof=/dev/vx/dsk/adg/avol \
  /dev/vx/dsk/adg/cacheavol /mnt_snap
```

Or, by using the original mount point (the result is identical):

```
# mount -F vxfs -o snapof=/mnt /dev/vx/dsk/adg/cacheavol /mnt_snap
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/avol 1048576   27989  956808     3%    /mnt
/dev/vx/dsk/adg/cacheavol
                     1048576   27989  956801     3%    /mnt_snap
# ls -lA /mnt*
/mnt:
total 20480
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found

/mnt_snap:
total 20480
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found
```

As you can see, the original file system and its associated snapshot exactly *look* like two independent file systems on the surface. Do they also *behave* like independent file systems? Let's play a little bit:

```
# mkfile 10m /mnt/file1.10m
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/avol 1048576   38229  947207     4%    /mnt
/dev/vx/dsk/adg/cacheavol
                     1048576   27989  956801     3%    /mnt_snap
```

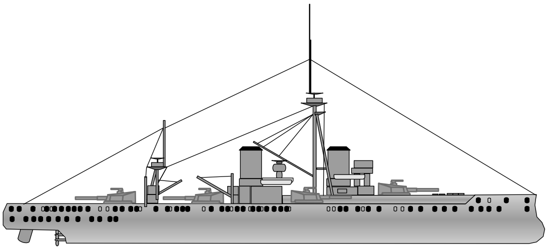
```
# ls -lA /mnt*
/mnt:
total 40960
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
-rw------T   1 root     root     10485760 Sep  6 18:07 file1.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found

/mnt_snap:
total 20480
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found
# rm /mnt/file0.10m
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/avol 1048576   27989  956808     3%    /mnt
/dev/vx/dsk/adg/cacheavol
                     1048576   27989  956801     3%    /mnt_snap
# ls -lA /mnt*
/mnt:
total 20480
-rw------T   1 root     root     10485760 Sep  6 18:07 file1.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found

/mnt_snap:
total 20480
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found
# rm /mnt_snap/file0.10m
rm: /mnt_snap/file0.10m: override protection 600 (yes/no)? yes
rm: /mnt_snap/file0.10m not removed: Read-only file system
# rm -f /mnt_snap/file0.10m
# ls -lA /mnt_snap
total 2048
-rw------T   1 root     root     10485760 Sep  6 18:04 file0.10m
drwxr-xr-x   2 root     root          96 Sep  6 17:57 lost+found
```

Ok, as long as the snapshot file system is accessed in read mode, it seems to behave like an independent file system (we will see another exception below). Write access is blocked (the override question is misleading, the "force" option when removing a file always suppresses STDERR).

Once again, we conclude with the main features of the VxFS snapshot. Compared with the former conclusion to the physical raw device snapshot, the ordinals do correspond.

1. The logical snapshot copy is physically dependent on the original file system, thus degrading application performance: snapshot read I/Os on unchanged data are read from the original file system, and write I/Os on still unmodified data on the original file system force a copy-on-first-write. Offhost processing is not possible.

2. The logical snapshot can only be accessed in read-only mode.
3. The logical snapshot can be used for immediate recovery only of corrupted application data, not in case of physically damaged devices (explained later).
4. The logical snapshot method is bound to VxFS.
5. The logical snapshot function is destroyed after an unmount of the snapshot file system, even more in case of a system crash.
6. The logical snapshot requires, compared to the original device, only a small portion of storage.
7. No preparatory data synchronization is necessary (instead copy-on-first-write after snapshot creation), the logical snapshot is available immediately.



The Full Battleship

9.3 Features of and Improvements on the Raw Device Snapshot

9.3.1 Snapshot Region Logging by the Data Change Log

In the Easy Sailing section, we just described the structure of a volume prepared for a raw device snapshot (especially the Data Change Object “DCO” and the Data Change Log Volume “DCL”). But we did not explain, why we need all these strange objects to perform a snapshot operation. Actually, the **vxsnap make** command would fail without those additional objects. But, on the other side, the "Technical Deep Dive" section will indeed show a quite simple procedure to create a snapshot based only on a current data plex within the volume, thus without any further objects, logs, and so on, neither as VxVM objects nor as kernel structures. So why all this complicated DC stuff?

An intelligent snapshot mechanism should provide an optimized framework to serve tasks more elaborate than simply creating a frozen copy, using it once and then deleting or forgetting it completely. Some examples should illustrate that:

A snapshot could be used regularly, e.g. on a daily basis for backup purposes. Indeed, we could delete today’s snapshot after having it used and recreate it completely from scratch tomorrow. But that would require full data synchronization every time the snapshot is created. Two major disadvantages readily come to mind: (1) the snapshot is never available immediately, and (2) we have an awful amount of unnecessary synchronization I/O degrading our system performance every time.

To approach the latter problem: why is synchronisation unnecessary? We could, physically spoken, skip an overwhelming portion of the synchronization, because most of our volume data did not change in the period between the previous and the current snapshot (the actual amount, of course, depends on the I/O behavior of the application). Currently, after having taken the previous snapshot, we do not have an appropriate object to log data changes. If the volume kept track of such changes, VxVM would know which regions to resynchronize and which to keep unmodified when “refreshing” the snapshot with the current data set.

This strongly desired log structure is represented by the Data Change Object (DCO) with its associated Data Change Log volume (DCL). The DCO links the application volume with its DCL volume providing some attributes concerning the features of the DCL. The most important attribute is called **regionsize** or **regionsz**, depending on the command line context. It defines the size of a contiguous region within the address space of the volume represented by one bit within the DCL volume.

The coded set of attributes shown by **vxprint** in its standard usage does not show the **regionsize** attribute. Therefore, we need special options to get its current value defining the bitmap structure of a snapshot "prepared" or a volume already "snapshot". Two examples, the first to use a comprehensible procedure, the second, deadly complicated, for scripting purposes (note, that **vxprint -e** needs the volume record ID **rid** to determine the associated DCO parent volume, not its name):

```
# vxprint -rLtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   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         c1t1d2   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-02     avol_dcl-02  adg01    2097152  544      0         c1t1d0   ENA
# vxprint -g adg -F %regionsz avol_dco
128
# vxprint -g adg -cF %regionsz -e dco_parent_vol=$(vxprint -g adg -F %rid avol)
128
```

The number 128 stands, as usual, for 128 sectors, that is 64 kB. So, one bit within the DCL bitmap represents 64 kB within its data volume (as we have seen in the commands above, this volume is also called DCO parent volume, while the DCL volume is never called DCO child volume). If any amount of data within such a region is modified, its corresponding bit is set, marking that region's need for resynchronization. Given our example parent volume with its size of 1 GB (which comprises 16,384 = 2^{14} regions of 64 kB), we need 16,384 bits or 2,048 bytes or 2 kB space to form the region bitmap. But surprisingly, the bitmap volume is much larger in size (544 sectors = 272 kB). Well, one reason is, that the DCL volume contains a multi-function bitmap of 33 levels providing not only improved snapshot characteristics (see the "Technical Deep Dive" part). Furthermore, we need some "global", region independent attribute data. There may be still further explanations, but they are unknown to us, they are not officially documented.

If, for any reason, the region size must be different from the default, you can specify it. We mention the procedure to achieve it not only in order to introduce a new keyword of **vxsnap**, but also to show an interesting error message concerning the multi-function bitmap (explained later). The "restore" example below demonstrates that, under special conditions, the resynchronization I/O size depends on the region size. And, what is more, we urgently need it when creating full sized instant snapshots (see below).

```
# vxsnap -g adg unprepare avol
VxVM vxassist ERROR V-5-1-6169  Volume avol has drl attach to it, use -f option
to remove drl
# vxsnap -g adg -f unprepare avol
# vxprint -rtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   ENA
# vxsnap -g adg prepare avol regionsize=32
# vxprint -rLtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   ENA
dc avol_dco     avol         avol_dcl

v  avol_dcl     -            ENABLED  ACTIVE   1120     SELECT    -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   1120     CONCAT    -        RW
sd adg03-01     avol_dcl-01  adg03    0        1120     0         c1t1d2   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   1120     CONCAT    -        RW
sd adg01-02     avol_dcl-02  adg01    2097152  1120     0         c1t1d0   ENA
# vxprint -g adg -cF %regionsz -e dco_parent_vol=$(vxprint -g adg -F %rid avol)
32
```

Note the increased size of the DCL volume, because every 16 kB region is now mapped! Unlike our example, you should consider to increase the region size to get larger "restore" I/O sizes. Note also, that the flexible architecture of the bitmap is too "difficult" for the legacy **vxdco** command, use **vxsnap** instead.

To avoid too many confusing details now, we come back to our main question (DRL attributes and log version will follow): How can I make use of this logging feature, which should help to dramatically reduce the amount of data synchronization in case of a snapshot refresh? I only need to use another new keyword of **vxsnap**. The following example assumes a snapshot "prepared" volume (**vxsnap prepare** and **vxsnap addmir** already issued) and is surrounded by **vxtrace** and **vxstat** commands to demonstrate the effect:

```
# vxsnap -g adg make source=avol/newvol=SNAP-avol/plex=avol-03
# vxtrace -g adg -d /tmp/vxtrace.dump -o dev &
[1]     19003
# dd if=/dev/zero of=/dev/vx/rdsk/adg/avol bs=1024k count=4
```

```
4+0 records in
4+0 records out
# kill %1
[1] + Terminated               vxtrace -g adg -d /tmp/vxtrace.dump -o dev &
# vxtrace -g adg -f /tmp/vxtrace.dump -o dev
11 START write vdev avol block 0 len 2048 concurrency 1 pid 19037
11 END write vdev avol op 11 block 0 len 2048 time 2
12 START write vdev avol block 2048 len 2048 concurrency 1 pid 19037
12 END write vdev avol op 12 block 2048 len 2048 time 1
13 START write vdev avol block 4096 len 2048 concurrency 1 pid 19037
13 END write vdev avol op 13 block 4096 len 2048 time 1
14 START write vdev avol block 6144 len 2048 concurrency 1 pid 19037
14 END write vdev avol op 14 block 6144 len 2048 time 1
# vxtrace -g adg -d /tmp/vxtrace.dump -o all &
[1]     19049
# vxstat -g adg -r
# vxsnap -g adg reattach SNAP-avol source=avol
# kill %1
[1] + Terminated               vxtrace -g adg -d /tmp/vxtrace.dump -o all &
# vxtrace -g adg -f /tmp/vxtrace.dump -o all | grep atomic
78 START atomic_copy vol avol op 79 block 0 len 2048 nsrc 32 ndest 1
78 END atomic_copy vol avol op 79 block 0 len 2048 time 2
86 START atomic_copy vol avol op 87 block 2048 len 2048 nsrc 32 ndest 1
86 END atomic_copy vol avol op 87 block 2048 len 2048 time 1
94 START atomic_copy vol avol op 95 block 4096 len 2048 nsrc 32 ndest 1
94 END atomic_copy vol avol op 95 block 4096 len 2048 time 1
102 START atomic_copy vol avol op 103 block 6144 len 2048 nsrc 32 ndest 1
102 END atomic_copy vol avol op 103 block 6144 len 2048 time 2
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol avol                 4      8192   12.0         0         0    0.0
vol avol_dcl             0         0    0.0         0         0    0.0
```

Indeed, it works! Instead of a full resynchronization, only those volume blocks are resynchronized which were previously overwritten by the **dd** command. It is, by the way, completely accidental that the I/O size of the **dd** command is identical to that of the resynchronization thread: 2,048 sectors = 1,024 kB = 1 MB. This is the Atomic Copy default, a snap "reattach" is indeed a plex attach, **vxtask list** would show the I/O type ATCOPY within the operation PLXSNAP. We simply have chosen 1 MB for **dd** to get corresponding numbers in both **vxtrace** outputs.

So we have solved one major disadvantage of a physical snapshot: Only data modified since the snapshot was taken are rewritten to the reattached snapshot plex. Not only is the amount of synchronization I/O dramatically reduced together with a lower system load. Furthermore, the plex marked for snapshot purposes becomes available for the next snapshot quite a lot faster. A few pages later, we will learn another procedure to really immediately bring the snapshot to the current data state (at least it looks and behaves so).

But now, we will first turn to another feature of the raw device snapshot.

9.3.2 Reverting the Resynchronization Direction

It should not happen, but it could happen that, while the snapshot still exists, the original device becomes unusable, either by hardware failures or by corrupted (application) data: lost files or database tables, invalid values, patches of the sort "hot destroy" instead of "hot fix". Note that volume redundancy does not protect against the latter scenario! Our best copy of volume data is most likely provided by the snapshot. Of course, in case of a disk outage, we need to recover disks and disk group first. Under normal conditions, the synchronization is directed by VxVM from the ENABLED/ACTIVE plexes (not by the volume layer) to the snapshot plex, which will become a member of the original volume once again. In this special case, we need a reversed synchonization direction: the original plexes with I/O fail or damaged data enter the STALE state (that's why the application access must be stopped first), the "snapbacked" plex forms the single current volume address space (at this stage, the application could already be restarted!), and finally, the synchronization thread is started from the volume based on the latter plex to the stale ones.

```
# vxsnap -g adg make source=avol/new=SNAP-avol/plex=avol-03
# dd if=/dev/zero of=/dev/vx/rdsk/adg/avol bs=1024k count=4
4+0 records in
4+0 records out
# vxstat -g adg -r
# vxsnap -g adg restore avol source=SNAP-avol destroy=yes
# vxstat -g adg -vp
                      OPERATIONS          BLOCKS           AVG TIME(ms)
TYP NAME              READ     WRITE      READ     WRITE   READ  WRITE
vol avol                 0         0         0         0    0.0    0.0
pl  avol-01              0        64         0      8192    0.0    8.0
pl  avol-02              0        64         0      8192    0.0    7.7
pl  avol-03             65         0      8208         0    2.2    0.0
vol avol_dcl             0         0         0         0    0.0    0.0
pl  avol_dcl-01         18        29       258       434    0.0    1.0
pl  avol_dcl-02          0        29         0       434    0.0    1.0
pl  avol_dcl-03          6         5        96        80    1.7    8.0
```

The testing scenario resembles the former one (we skipped **vxtrace**, its output would be too long). Five remarks:

1. The **source** keyword in the latter **vxsnap** command does not indicate the original volume, but the snapshot volume, thus specifying the synchronization/"restore" direction.
2. Physical synchronization may be omitted by adding **syncing=no**, the application volume would be restored "logically" (see logical snapshots below for further details).
3. Without **destroy=yes** (or with **destroy=no**), the snapshot volume would remain a separate volume (this is the reverted equivalent to **vxsnap refresh** explained

below).

4. The I/O sizes are smaller compared to a **reattach** resynchronization: 8,192 sectors = 4,096 kB in 64 I/Os correspond an I/O size of 64 kB, which is our default DCO region size. **vxtask list** would show the I/O type SNAPSYNC within the operation SNAPSYNC. We have smaller granularity for resynchronization, but the main I/O strategy remains identical.
5. Don't ask us, why the source plex was read one I/O in addition, we do not know the answer.

9.3.3 The Snap Objects

Another new object type related to snapshots needs further investigation, the snap object ("sp") linking the snapshot volume to its original volume and vice-versa. Why do we need them? The first observation: In case of the keywords **reattach**, **restore**, and **refresh**, the command **vxsnap** would fail without the **source** volume keyword and a specified (target) volume. The second observation, seemingly contradictory: We will demonstrate a procedure to instantly create a snapshot relation between previously independent volumes a few pages later. To conclude, the snap objects mark the volumes as members of a snapshot interconnection (called "chain"), thus prohibiting their inadvertently snap **unprepare** or volume destruction:

```
# vxsnap -g adg unprepare SNAP-avol
VxVM vxsnap ERROR V-5-1-6170  Volume SNAP-avol is in snapshot chain
# vxassist -g adg remove volume SNAP-avol
VxVM vxassist ERROR V-5-1-10127 deleting volume SNAP-avol:
        Record is associated
```

Warning: The snap objects do not protect against the **vxedit -rf rm** command, in spite of the manual page to **vxsnap dis**! The snapshot volume would be destroyed together with all snap objects, leaving the original volume in the snap "prepared" state (and vice-versa).

But the most important function of the snap objects is to indicate, that the intelligent DC log is ready for use in case of snapshot **reattach**, **restore**, and **refresh**. Without snap objects, it is possible to create or recreate a snapshot relation between the two data volumes (full sized instant snapshot), but any synchronization task would mean 100 percent synchronization.

How does the snap object identify its source and its target volume? The **vxprint -t** command does not show any appropriate attributes, only the location of the **sp** object under the DCL volume, and the naming convention indicates source and target (which is not a must, as we know). Other options (**-l**, **-A**, **-a**, **-m**) print two snap object attributes: GUIDs unmistakably identifying the source (attribute name "vol_guid") and the target volume ("snapshot_vol_guid"). The **-F** option allows to specify a desired output format, as given in the following example:

```
# vxprint -g adg -cF 'snapobject %name: source=%vol_guid '\
  'target=%snapshot_vol_guid'
object SNAP-avol_dco: source=- target=-
object avol_dco: source=- target=-
object SNAP-avol_snp: source={71840bae-1dd2-11b2-88f6-0003ba07fc88}
target={78b99146-1dd2-11b2-88ed-0003ba07fc88}
object avol_snp: source={78b99146-1dd2-11b2-88ed-0003ba07fc88} target={71840bae-
1dd2-11b2-88f6-0003ba07fc88}
```

Note: The first two lines of the output belong to the data change objects linking the data volumes to their DCL volume. The option **-c** cannot differentiate between snap objects and DC objects. Note also, that the literal expression "snapobject" at the beginning of the argument to the **-F** option was shortened to "object" as a result of an internal programming error of **vxprint**.

Well, the identification by GUIDs indeed is unique, but it is quite unreadable for us. The Shell with its powerful capabilities (here: loops, conditionals, command substitution) allows us to generate a quite unreadable expression, but the output is of the sort we like to see:

```
# printf '%-15s %-15s %s\n' SNAP_OBJECT SOURCE TARGET; \
  vxprint -g adg -cF '%type %name %vol_guid %snapshot_vol_guid' |
  while read Type Name VGuid SVGuid; do
  [[ $Type == sp ]] || continue
  printf '%-15s %-15s %s\n' $Name \
  $(vxprint -g adg -vne v_guid=$VGuid) \
  $(vxprint -g adg -vne v_guid=$SVGuid)
  done
SNAP_OBJECT     SOURCE          TARGET
SNAP-avol_snp   avol            SNAP-avol
avol_snp        SNAP-avol       avol
```

Fortunately, **vxsnap** itself provides a powerful keyword to print snapshot information. We show two examples:

```
# vxsnap -g adg print
NAME    SNAPOBJECT    TYPE    PARENT    SNAPSHOT    %DIRTY    %VALID

avol     --           volume   --       --           --       100.00
        SNAP-avol_snp  volume   --       SNAP-avol    0.00     --

SNAP-avol avol_snp     volume   avol     --           0.00     100.00
```

The relation of the snap objects to the source and target volumes is printed together with "dirty" and "valid" percentage (explained later).

```
# vxsnap -g adg -n print
NAME          DG         OBJTYPE SNAPTYPE PARENT     PARENTDG   SNAPDATE
```

```
avol          adg        vol     -        -          -          - -
SNAP-avol     adg        vol     mirbrk   avol       adg        2008/09/14 08:54
```

This command does not show the names of the snap objects, but, besides the relation of original and snapshot volume, the snapshot type ("mirror break", we will learn another type later) and, quite important, the snapshot date, i.e. the date the snapshot plex was dissociated from the original volume.

The keyword **list** of **vxsnap** produces nearly the same output and may be skipped for further investigation.

9.3.4 Clearing the Snapshot Relation

Sometimes you could decide to never again bring back the snapshot volume to its original location, e.g. you want to go on with your application in the test location for an undefined period. It would simplify the administration to cut off the snapshot interconnection:

```
# vxprint -rLtg adg
[…]
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  ROUND     -        fsgen
pl avol-03      SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-02     avol-03      adg03    544      2097152  0         c1t1d2   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 adg02-02     avol_dcl-03  adg02    2097152  544      0         c1t1d1   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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   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         c1t1d2   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-02     avol_dcl-02  adg01    2097152  544      0         c1t1d0   ENA
sp SNAP-avol_snp avol        avol_dco

# vxsnap -g adg dis SNAP-avol
# vxprint -rLtg adg
[…]
```

```
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  ROUND     -        fsgen
pl avol-03      SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-02     avol-03      adg03    544      2097152  0         c1t1d2   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 adg02-02     avol_dcl-03  adg02    2097152  544      0         c1t1d1   ENA


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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   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         c1t1d2   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-02     avol_dcl-02  adg01    2097152  544      0         c1t1d0   ENA
```

The snap objects are removed. From now on, VxVM handles both volumes as completely distinct volumes, even though they are still snap "prepared". A somewhat softer version is performed by **vxsnap split**: In case of a still running synchronization thread of a full sized instant snapshot (see below), it would fail. This keyword is designed to temporarily remove the snap objects only for a fully synchronized snapshot and to recreate them at any time by way of building a logical snapshot (see below).

9.3.5 Deleting the Snapshot

Combined with our knowledge about volume destruction, we are now able to "cleanly" remove a snapshot (don't forget to stop application access and/or to unmount the corresponding file system as the first step):

```
# vxsnap -g adg split SNAP-avol
# vxassist -g adg remove volume SNAP-avol
```

The procedure for impatient and courageous guys never committing mistakes:

```
# vxedit -g adg -rf rm SNAP-avol
```

9.3.6 Offhost Processing

A physical snapshot is a frozen, but nevertheless complete copy of the volume address space. As already mentioned in the introduction to snapshots, we could want to transfer the access to this copy to another host, e.g. for the following purposes: offhost backup, exhaustive reporting with several data warehouse like full table scans, separated testing environment, and so on.

But alas! VxVM regulates the access to its volumes on a per disk group base. Unfortunately, the original volume and its snapshot volume are kept in the same disk group. We must conclude, that either offhost processing is impossible or we need the expensive Cluster Volume Manager license to enable parallel access to disk groups or the disk group must be split. The latter is indeed implemented.

The Disk Group Split and Join feature (DGSJ) was introduced in VxVM 3.2 and got an improved administration by the **vxsnap** command. Splitting a disk group into two completely independent disk groups requires some intelligent planning of storage allocation of the volumes. So we usually need to specify the storage attributes when preparing the volume for snapshots and creating the snapshot related objects.

A standard volume is not bound to build its address spaces (plexes) from specific storage, the subdisk is a arbitrarily configurable instance between the physical and the virtual layer, in other words, its virtual position within the plex is completely independent from its physical position on the disk. Nevertheless, **vxassist** has a reasonable built-in limitation to serve redundancy and performance needs: You cannot stripe or mirror over subdisks on the same disk as long as the subdisks are part of the top level or the same sublevel volume.

When splitting a complex snapshot structure into two different disk groups, we do not want to destroy structures we want to keep alive and to go on working properly (the original volume should remain online). We do not want, as an example, to destroy its twofold redundancy (two data plexes). Since it is impossible and indeed not suitable for integrity needs to keep the original access of host A to mirror 1 and to switch the access to mirror 2 to host B, while the volume fully remains in use, VxVM does not allow to rupture a volume by splitting the disk group. Therefore, all disks used by a volume must either remain in the original disk group or completely split off into the new disk group.

The DCL volume of a snapshot "prepared" application volume is an integral part of its DCO parent volume, associated by the DC object. The disk group split must not destroy this logging volume as well, and it must not cut off its logging relation to the parent volume. The same is true for the snapshot side of our volume structure: The snapshot volume and its DCL volume if carrying redundancy (which is not the default) need to be kept connected.

To sum up: The set of disks used to build the application volume and its DCL volume on the "left" side (see image below) and the set of the snapshot volume and its DCL volume on the "right" side need to be strictly exclusive. Furthermore, all other volumes or comparable associations (replicated volume groups, volume sets, DCO logs, cache subdisks), if there are any within the same disk group, must conform to that rule. Otherwise, our attempt to split the disk group will fail! The next example (this time with the subdisks drawn) shows a properly configured scenario: All subdisks of the original volume and its DCL volume reside on disks **adg01** and **adg02**, while the snapshot part only uses disk adg03.

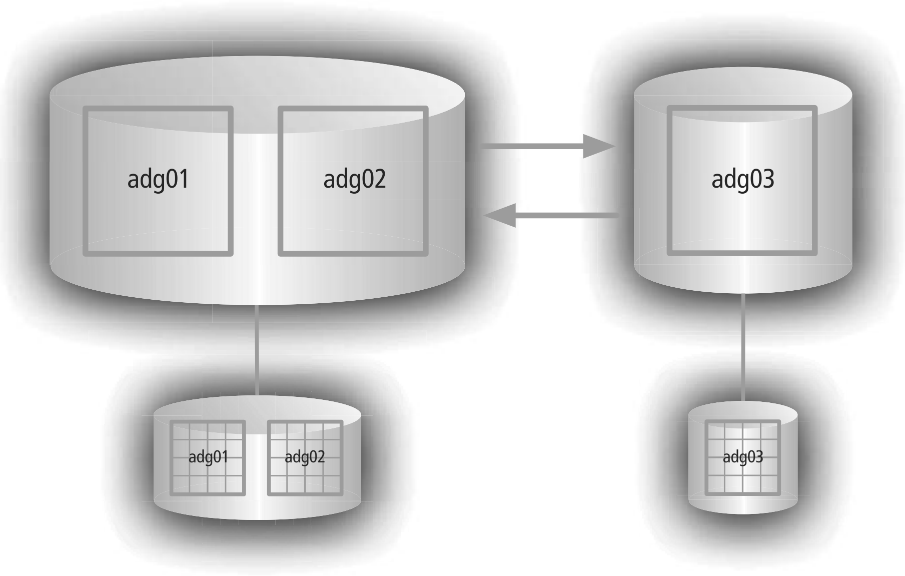


Figure 9-6: Application and snapshot volume ready for disk group split

But what can be done to achieve this layout? Make use of the storage attributes when creating volume and snapshot objects:

```
# vxassist -g adg make avol 1g layout=mirror nmirror=2 init=active \
  alloc=adg01,adg02
# vxsnap -g adg prepare avol alloc=adg01,adg02
# vxsnap -g adg addmir avol alloc=adg03
# vxprint -rLtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   ENA
pl avol-03      avol         ENABLED  SNAPDONE 2097152  CONCAT    -        WO
sd adg03-01     avol-03      adg03    0        2097152  0         c1t1d2   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 adg01-02     avol_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg02-02     avol_dcl-02  adg02    2097152  544      0         c1t1d1   ENA
```

```
pl avol_dcl-03  avol_dcl     DISABLED DCOSNP   544      CONCAT    -        RW
sd adg03-02     avol_dcl-03  adg03    2097152  544      0         c1t1d2   ENA
```

And what to do, if the volumes already exist and cannot be removed and recreated, because they are in use? Well, we could move the subdisks to the desired locations using **vxsd mv** or **vxassist move**, sensibly on the DCL volume due to its small size. Sometimes, there is no other way to free a disk from concurrent use by different volumes. In our single application volume scenario, we would like to introduce another way, mostly easier to handle and sometimes useful for other purposes: We simply switch the snap markers to the appropriate plexes. Switching is done by removing the marker from one plex and setting it to another one. Since it does not work exactly this way, we first provide three operation mode examples:

1. Switching the snap marker only for the data plex (the first command will remove the DCOSNP plex in the DCL volume, the latter will NOT recreate it):

```
# vxplex -g adg convert state=ACTIVE <snapdone-plex>
# vxplex -g adg convert state=SNAPDONE <active-plex>
```

2. Switching the snap marker to ACTIVE only for the DCL plex (you cannot revert it to DCOSNP):

```
# vxplex -g adg convert state=ACTIVE <dcosnp-plex>
```

3. Switching the snap markers for both, the data and the DCL plex (the first command will remove the DCOSNP plex in the DCL volume):

```
# vxplex -g adg -o dcoplex=<dcosnp-plex> convert state=ACTIVE <snapdone-plex>
# vxplex -g adg -o dcoplex=<active-plex> convert state=SNAPDONE <active-plex>
```

Due to the removal of DCOSNP plexes when converting the appropriate data plex to the active state, we conclude that we must recreate the lost DCL plex before switching both plexes to serve as snapshot plexes:

```
# vxplex -g adg convert state=ACTIVE <snapdone-plex>
# vxassist -g adg mirror <dcl-volume> [alloc=<disk>]
# vxplex -g adg -o dcoplex=<active-plex> convert state=SNAPDONE <active-plex>
```

Having successfully prepared our subdisk usage, we perform the disk group split. There is no risk in executing the following command, because it will fail instead of destroying related object associations or making volumes in use inaccessible by moving them into a different disk group:

```
# vxsnap -g adg make source=avol/new=SNAP-avol/plex=avol-03
# vxdg split adg offdg SNAP-avol
# vxdisk list
```

```
[…]
c1t1d0s2     auto:cdsdisk    adg01        adg          online
c1t1d1s2     auto:cdsdisk    adg02        adg          online
c1t1d2s2     auto:cdsdisk    adg03        offdg        online
# vxprint -rLtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   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 adg01-02     avol_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg02-02     avol_dcl-02  adg02    2097152  544      0         c1t1d1   ENA
sp SNAP-avol_snp avol        avol_dco
# vxprint -rLtg offdg
[…]
v  SNAP-avol    -            DISABLED ACTIVE   2097152  ROUND     -        fsgen
pl avol-03      SNAP-avol    DISABLED ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     avol-03      adg03    0        2097152  0         c1t1d2   ENA
dc SNAP-avol_dco SNAP-avol   SNAP-avol_dcl

v  SNAP-avol_dcl -           DISABLED ACTIVE   544      ROUND     -        gen
pl avol_dcl-03  SNAP-avol_dcl DISABLED ACTIVE  544      CONCAT    -        RW
sd adg03-02     avol_dcl-03  adg03    2097152  544      0         c1t1d2   ENA
sp avol_snp     SNAP-avol    SNAP-avol_dco
```

Now, the new disk group containing the snapshot volume and its DCL volume is ready for offhost processing. We are able to deport it and import it on another host, start the volumes, and attend our offhost duties. In many cases, it is quite reasonable to revert this procedure to be prepared for the next snapshot. First, we must stop our offhost processing, then deport the disk group and import it once again on the original host. Below are the steps required to join the already imported offhost disk group with the application disk group, start the snapshot volume and its DCL volume affected by the volume move (option **-m**), and reattach them to their original volumes (a **refresh** or a **restore** operation would only modify the keyword of the last command):

```
# vxdg join offdg adg
# vxrecover -g adg -m
# vxsnap -g adg reattach SNAP-avol source=avol
```

9.3.7 Full Sized Volume Based Instant Snapshots

Let's turn to another functionality of the multi-layered DCL bitmap! We already mentioned, that the amount of time needed to synchronize a new snapshot plex or to bring an existing snapshot volume to the current state of application data is somewhat harmful. Sometimes, we immediately need the snapshot.

One layer within the DCL bitmap of the snapshot volume provides pointer functionality: If the bit is set, its correspondent region data are physically stored in the snapshot volume itself, whether these data are the original snapshot data or data modified by write access to the snapshot volume. If the bit is cleared, its correspondent region data are read from the original device, because data did not change since the snapshot. This kind of procedure to simulate a physical snapshot is called "logical snapshot".

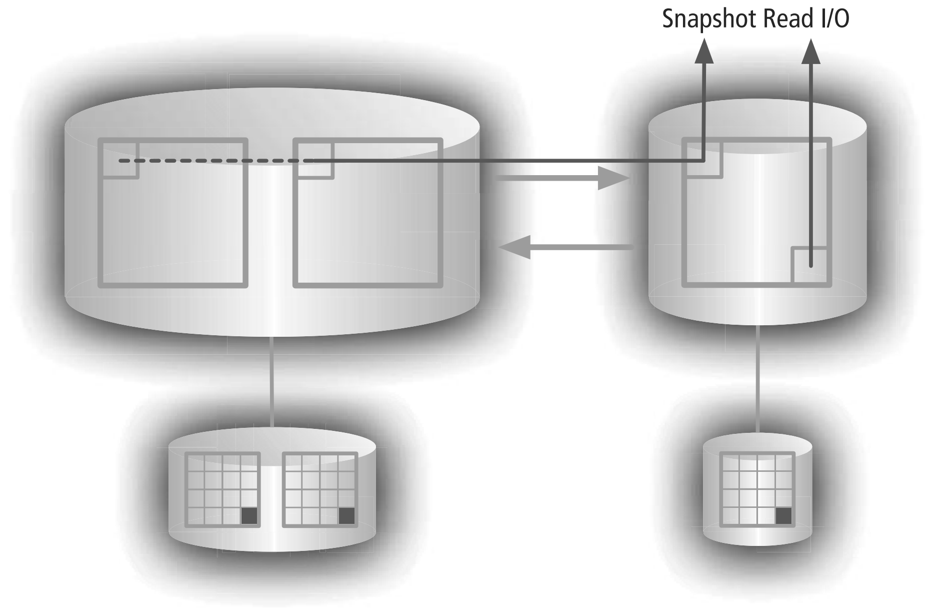


Figure 9-7: Read access to a full sized "logical" snapshot

Such a snapshot is indeed immediately ready for use. We only need to specify an appropriate volume as a snapshot for the application volume, VxVM will clear all bits within the "logical snapshot" bitmap, thus providing a simulated copy of the application volume accessed by another volume driver. We will now explain the mode of operation together with the necessary configuration step by step. Let's start at the very beginning with the creation of the application volume and the volume to become its logical snapshot. Note that the size of both top-level volumes and the region size of both bitmaps are identical.

```
# vxassist -g adg make avol 1g layout=mirror init=active alloc=adg01,adg02
# vxsnap -g adg prepare avol alloc=adg01,adg02
# vxprint -rLtg 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         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1   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 adg01-02     avol_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
```

```
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -         RW
sd adg02-02     avol_dcl-02  adg02    2097152  544      0         c1t1d1    ENA
# vxassist -g adg make SNAP-avol 1g alloc=adg03
# vxprint -g adg -F %regionsz avol_dco
128
# vxsnap -g adg prepare SNAP-avol regionsize=128 alloc=adg03
# vxprint -rLtg adg
[…]
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  SELECT    -         fsgen
pl SNAP-avol-01 SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT    -         RW
sd adg03-01     SNAP-avol-01 adg03    0        2097152  0         c1t1d2    ENA
dc SNAP-avol_dco SNAP-avol   SNAP-avol_dcl

v  SNAP-avol_dcl -           ENABLED  ACTIVE   544      SELECT    -         gen
pl SNAP-avol_dcl-01 SNAP-avol_dcl ENABLED ACTIVE 544    CONCAT    -         RW
sd adg03-02     SNAP-avol_dcl-01 adg03 2097152 544      0         c1t1d2    ENA


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         c1t1d0    ENA
pl avol-02      avol         ENABLED  ACTIVE   2097152  CONCAT    -         RW
sd adg02-01     avol-02      adg02    0        2097152  0         c1t1d1    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 adg01-02     avol_dcl-01  adg01    2097152  544      0         c1t1d0    ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -         RW
sd adg02-02     avol_dcl-02  adg02    2097152  544      0         c1t1d1    ENA
```

The current content of the disk group exactly looks like an application volume with its split or dissociated snapshot volume (the snap objects are missing). But remember: Until now, our volumes never had a snapshot relation. And keep in mind: Creating the volume designed to serve as snapshot took only a few seconds (unless you are not familiar with the procedure).

The next step is to tell VxVM that the latter volume should serve as a logical snapshot to the application volume. Quite easy with the **vxsnap** command:

```
# vxsnap -g adg make source=avol/snap=SNAP-avol sync=no
# vxprint -rLtg adg
…
v  SNAP-avol    -            ENABLED  ACTIVE   2097152  SELECT    -         fsgen
pl SNAP-avol-01 SNAP-avol    ENABLED  ACTIVE   2097152  CONCAT    -         RW
sd adg03-01     SNAP-avol-01 adg03    0        2097152  0         c1t1d2    ENA
dc SNAP-avol_dco SNAP-avol   SNAP-avol_dcl
```

```
v  SNAP-avol_dcl -            ENABLED  ACTIVE   544      SELECT    -        gen
pl SNAP-avol_dcl-01 SNAP-avol_dcl ENABLED ACTIVE 544     CONCAT    -        RW
sd adg03-02      SNAP-avol_dcl-01 adg03 2097152 544      0         c1t1d2   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         c1t1d0   ENA
pl avol-02       avol         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01      avol-02      adg02    0        2097152  0         c1t1d1   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 adg01-02      avol_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
pl avol_dcl-02   avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg02-02      avol_dcl-02  adg02    2097152  544      0         c1t1d1   ENA
sp SNAP-avol_snp avol         avol_dco
```

We already recognize the snap objects linking the snapshot volume to its application volume and vice-versa. Currently, assuming no application write I/Os, the snapshot bitmap of **SNAP-avol_dcl** marks all snapshot regions as "invalid", i.e. all data must be read from the application volume.

The command **vxsnap** provides a useful keyword to print the amount of "valid", i.e. to the snapshot volume already synchronized regions:

```
# vxsnap -g adg print
NAME     SNAPOBJECT    TYPE     PARENT    SNAPSHOT    %DIRTY    %VALID

avol      --           volume    --        --           --       100.00
        SNAP-avol_snp   volume    --        SNAP-avol    0.00      --

SNAP-avol avol_snp      volume    avol      --           0.00      0.00
```

The last word (column **%VALID**) in the last line (object name **SNAP-avol**) shows, that no data were already stored on the snapshot volume (**0.00%**). But what becomes of data overwritten by write access of the application? The snapshot mechanism must store the original data set to the snapshot volume before it is physically overwritten by the application for the first time ("copy on first write"). Furthermore, the corresponding bit in the DCL bitmap needs to be set to indicate that the snapshot is prohibited to read those region data from the application volume.

Indeed, **vxsnap** prints out, that some portions of the snapshot volume (depending on the application I/O size) are now "valid", i.e. physically stored within the snapshot volume. We overwrite the first 100 MB of our volume:

```
# dd if=/dev/zero of=/dev/vx/rdsk/adg/avol bs=1024k count=100
100+0 records in
100+0 records out
# vxsnap -g adg print
NAME     SNAPOBJECT     TYPE     PARENT     SNAPSHOT     %DIRTY     %VALID

avol      --            volume    --         --            --        100.00
        SNAP-avol_snp  volume    --        SNAP-avol     9.77       --

SNAP-avol avol_snp      volume    avol      --             9.77      9.77
```

Don't forget: 100 MB is less than 10% of 1 GB, because 1 GB consists of 1,024 MB! 9.77% of the original volume is already copied to the snapshot volume, and both volumes differ in 9.77% of data (column **%DIRTY**).

We could want to convert the logical snapshot into a physical one, e.g. to enable offhost processing or to save copy-on-first-write I/Os at a later, busy period by writing application volume data to the snapshot volume right now. We could, of course, show real patience until all regions of the application volume are overwritten by new data. But there are two ways to start immediate snapshot synchronization.

1. At any time, we can start data transfer to the snapshot volume, enabling, if desired, the "background" operation mode (option **-b**) and a performance throttle (specified in milliseconds). We may also pause and resume it with the throttle started by, or completely terminate it.

```
# vxsnap -g adg -b [-o slow=<#>] syncstart SNAP-avol
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   172         SNAPSYNC/R 10.06% 0/2097152/210944 SNAPSYNC SNAP-avol adg
# vxsnap -g adg syncpause SNAP-avol
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   172         SNAPSYNC/P 11.23% 0/2097152/235520 SNAPSYNC SNAP-avol adg
# vxsnap -g adg print
NAME     SNAPOBJECT     TYPE     PARENT     SNAPSHOT     %DIRTY     %VALID

avol      --            volume    --         --            --        100.00
        SNAP-avol_snp  volume    --        SNAP-avol     9.77       --

SNAP-avol avol_snp      volume    avol      --             9.77      11.33
# vxsnap -g adg syncresume SNAP-avol
# vxtask list
TASKID  PTID TYPE/STATE    PCT   PROGRESS
   172         SNAPSYNC/R 11.52% 0/2097152/241664 SNAPSYNC SNAP-avol adg
# vxsnap -g adg syncstop SNAP-avol
# vxsnap -g adg print
```

```
NAME     SNAPOBJECT     TYPE     PARENT     SNAPSHOT     %DIRTY     %VALID

avol      --            volume    --        --            --        100.00
        SNAP-avol_snp  volume    --        SNAP-avol    9.77      --

SNAP-avol avol_snp     volume    avol      --           9.77      13.38
# vxsnap -g adg syncresume SNAP-avol
VxVM vxsnap ERROR V-5-1-6680 No instant operation is running on the volume SNAP-
avol
```

2. When creating the snapshot relation between the two volumes, we may simply omit the keyword **sync** or write **sync=yes**. This will immediately start a synchronization thread on all volume regions:

```
# vxsnap -g adg make source=avol/snap=SNAP-avol [sync=yes]
```

A fully synchronized snapshot volume does not only look and behave like a physical snapshot, it is actually a physical snapshot, and there is no difference in the result compared to the legacy snapshot mechanism: all snapshot I/Os are taken from the snapshot device, offhost processing is possible, and so on.

Another remark concerning full sized instant snapshots: The volume intended to become the instant snapshot of an application volume may not reside within the same disk group. When establishing the snapshot relation, we may specify within the slash separated tuple of the **vxsnap** command the keyword **snapdg**:

```
# vxsnap -g adg make source=avol/snap=SNAP-avol/snapdg=offdg
```

9.3.8 Snapshot Refresh

Now, with the knowledge of logical snapshot relations based on the multi-functional bitmap of the DCL volume, we will easily understand another feature of the DCO based raw device snapshots, whether in complete or partial physical state: the snapshot refresh. "Refreshing" the snapshot, that is updating the data set represented by the snapshot to the current content of the application volume, simply means converting the snapshot DCO bitmap from its current state (most probably a mixture of physical and logical pointer bits or already indicating a fully synchronized snapshot) to a plain logical bitmap.

At any time, independent from the procedure which created the snapshot volume, but nevertheless only without current access to it, a snapshot volume can be refreshed. The refresh operation may invoke background synchronization at the same time (default behavior), but this is, compared to the logical snapshot creation, just as well optional.

```
# vxsnap -g adg refresh SNAP-avol [sync=no]
```

9.3.9 Space Optimized Volume Based Instant Snapshots

Consider you do not want or need a physical snapshot at all, and your snapshot will be used only for a few hours (e.g. for backup purposes). Another physical instance of the volume address space, as required by the volume snapshot mechanisms hitherto explained, could evoke inconvenient questions about wasting storage. And those questions should be taken seriously, because they point to an undeniable weakness of physical snapshots: Data remaining unchanged during the period of the snapshot are stored twice (application and snapshot volume, if physical snapshot) or waste space on the snapshot volume (logical snapshot). For logical snapshots, it would be sufficient to provide storage only for the original data, before they are overwritten by the application. Data unchanged remain stored on the application volume, while the snapshot bitmap simply continue to point to them.

Maybe you remember the construct of the VxFS based logical snapshot presented in the "Easy Sailing" section. Indeed, we created a snapshot device containing a bitmap of exactly that functioning and providing the storage required to save the original data, before they were overwritten. We mentioned, that for many temporary purposes 10% of the application size would be sufficient to serve as a snapshot device.

The VxVM based space optimized snapshot uses a somewhat different architecture in order to support a shared cache, i.e. a cache providing dynamic storage for more than one application volume. Thus, several application volumes can store their original data in one storage device to benefit from dynamic storage requirements: an application requests for snapshot purposes more storage, another application less than expected.

But what is our snapshot device now? We talked about application and cache volumes, not about snapshot volumes. Well, the snapshot is indeed not a physical device anymore except for the small storage needed to build the already known DCO bitmap, marking whether the snapshot data are physically to be read from the application volume or from the cache volume. The snapshot volume is still a regular volume as well as its plex, but the subdisk is a virtual one (called "subcache"), not defined on a disk device or a subvolume (other subdisks are still not drawn in the following picture).

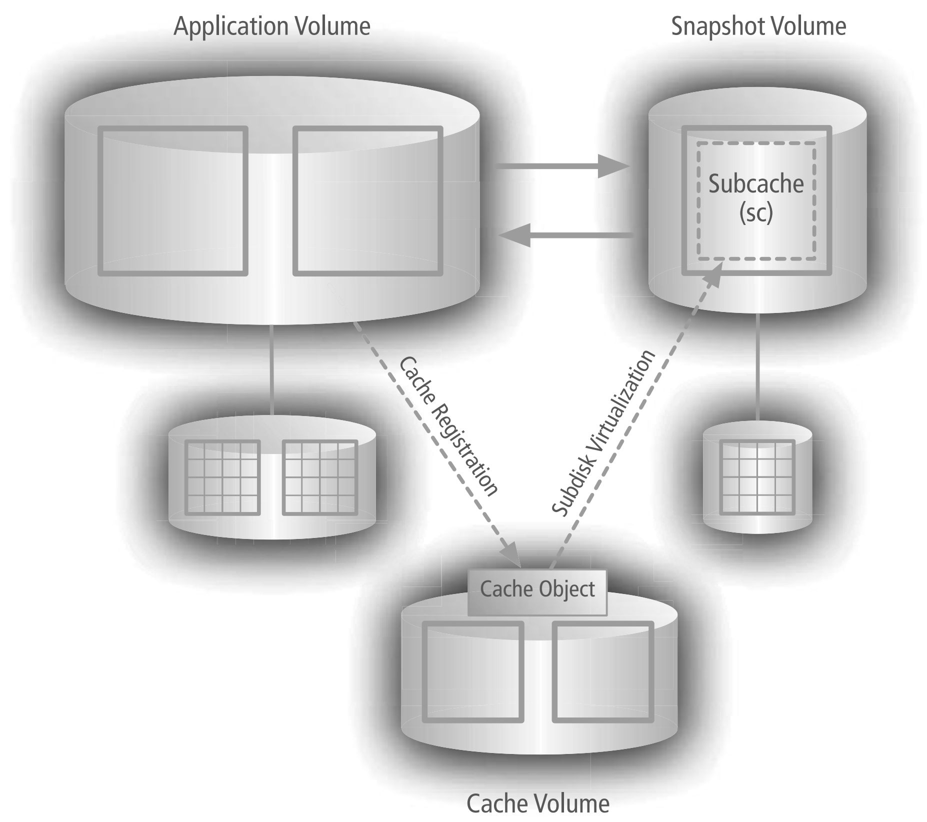


Figure 9-8: Space optimized snapshot with cache volume and subcache

Further details of space optimized snapshots with a shared cache volume are best demonstrated by the procedure to create them. We choose two application volumes in the simplest plex layout, **vol1** and **vol2** respectively.

```
# vxassist -g adg make vol1 1g layout=mirror nmirror=2 init=active \
  alloc=adg01,adg02
# vxassist -g adg make vol2 1g layout=mirror nmirror=2 init=active \
  alloc=adg03,adg04
# vxsnap -g adg prepare vol1 alloc=adg01,adg02
# vxsnap -g adg prepare vol2 alloc=adg03,adg04
```

Our cache volume will be mirrored to provide the same redundancy for the snapshots as for the application volumes. Its size of 256 MB allows an average of more than 10% of modified original data for both application volumes. Finally, we need a new object type called "cache object" serving as a cache volume registration instance for the snapshots and as a so-called in-core bitmap on used regions in the cache volume. For recovery purposes, the cache object can be started and stopped.

```
# vxassist -g adg make cvol 256m layout=mirror nmirror=2 init=active \
  alloc=adg05,adg06
# vxmake -g adg cache cobjcvol cachevolname=cvol
# vxcache -g adg start cobjcvol
# vxprint -rLtg adg
[…]
v  vol1         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol1-01      vol1         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol1-01      adg01    0        2097152  0         c1t1d0   ENA
pl vol1-02      vol1         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol1-02      adg02    0        2097152  0         c1t1d1   ENA
dc vol1_dco     vol1         vol1_dcl

v  vol1_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl vol1_dcl-01  vol1_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-02     vol1_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
pl vol1_dcl-02  vol1_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg02-02     vol1_dcl-02  adg02    2097152  544      0         c1t1d1   ENA


v  vol2         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol2-01      vol2         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol2-01      adg03    0        2097152  0         c1t1d2   ENA
pl vol2-02      vol2         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg04-01     vol2-02      adg04    0        2097152  0         c1t1d3   ENA
dc vol2_dco     vol2         vol2_dcl

v  vol2_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl vol2_dcl-01  vol2_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg03-02     vol2_dcl-01  adg03    2097152  544      0         c1t1d2   ENA
pl vol2_dcl-02  vol2_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg04-02     vol2_dcl-02  adg04    2097152  544      0         c1t1d3   ENA

co cobjcvol     cvol         ENABLED  ACTIVE

v  cvol         cobjcvol     ENABLED  ACTIVE   524288   SELECT    -        fsgen
pl cvol-01      cvol         ENABLED  ACTIVE   524288   CONCAT    -        RW
sd adg05-01     cvol-01      adg05    0        524288   0         c1t1d4   ENA
pl cvol-02      cvol         ENABLED  ACTIVE   524288   CONCAT    -        RW
sd adg06-01     cvol-02      adg06    0        524288   0         c1t1d5   ENA
```

Well, life is not always as easy as one could desire it! But it will get even more complicated, because we still have no snapshots.

```
# vxsnap -g adg make source=vol1/new=SNAP-vol1/cache=cobjcvol
# vxsnap -g adg make source=vol2/new=SNAP-vol2/cache=cobjcvol
# vxprint -rLtg adg
[…]
v  SNAP-vol1    -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl SNAP-vol1-P01 SNAP-vol1   ENABLED  ACTIVE   2097152  CONCAT    -        RW
sc SNAP-vol1-S01 SNAP-vol1-P01 cobjcvol 0      2097152  0         -        ENA
dc SNAP-vol1_dco SNAP-vol1   SNAP-vol1_dcl

v  SNAP-vol1_dcl -           ENABLED  ACTIVE   544      SELECT    -        gen
pl SNAP-vol1_dcl-01 SNAP-vol1_dcl ENABLED ACTIVE 544    CONCAT    -        RW
sd adg07-01     SNAP-vol1_dcl-01 adg07 0       544      0         c1t1d6   ENA
sp vol1_snp     SNAP-vol1    SNAP-vol1_dco


v  SNAP-vol2    -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl SNAP-vol2-P01 SNAP-vol2   ENABLED  ACTIVE   2097152  CONCAT    -        RW
sc SNAP-vol2-S01 SNAP-vol2-P01 cobjcvol 0      2097152  0         -        ENA
dc SNAP-vol2_dco SNAP-vol2   SNAP-vol2_dcl

v  SNAP-vol2_dcl -           ENABLED  ACTIVE   544      SELECT    -        gen
pl SNAP-vol2_dcl-01 SNAP-vol2_dcl ENABLED ACTIVE 544    CONCAT    -        RW
sd adg08-01     SNAP-vol2_dcl-01 adg08 0       544      0         c1t1d7   ENA
sp vol2_snp     SNAP-vol2    SNAP-vol2_dco


v  vol1         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol1-01      vol1         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol1-01      adg01    0        2097152  0         c1t1d0   ENA
pl vol1-02      vol1         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol1-02      adg02    0        2097152  0         c1t1d1   ENA
dc vol1_dco     vol1         vol1_dcl

v  vol1_dcl     -            ENABLED  ACTIVE   544      SELECT    -        gen
pl vol1_dcl-01  vol1_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-02     vol1_dcl-01  adg01    2097152  544      0         c1t1d0   ENA
pl vol1_dcl-02  vol1_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg02-02     vol1_dcl-02  adg02    2097152  544      0         c1t1d1   ENA
sp SNAP-vol1_snp vol1        vol1_dco


v  vol2         -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol2-01      vol2         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol2-01      adg03    0        2097152  0         c1t1d2   ENA
pl vol2-02      vol2         ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg04-01     vol2-02      adg04    0        2097152  0         c1t1d3   ENA
dc vol2_dco     vol2         vol2_dcl
```

```
v  vol2_dcl     -            ENABLED  ACTIVE   544      SELECT   -        gen
pl vol2_dcl-01  vol2_dcl     ENABLED  ACTIVE   544      CONCAT   -        RW
sd adg03-02     vol2_dcl-01  adg03    2097152  544      0        c1t1d2   ENA
pl vol2_dcl-02  vol2_dcl     ENABLED  ACTIVE   544      CONCAT   -        RW
sd adg04-02     vol2_dcl-02  adg04    2097152  544      0        c1t1d3   ENA
sp SNAP-vol2_snp vol2        vol2_dco

co cobjcvol     cvol         ENABLED  ACTIVE

v  cvol         cobjcvol     ENABLED  ACTIVE   524288   SELECT   -        fsgen
pl cvol-01      cvol         ENABLED  ACTIVE   524288   CONCAT   -        RW
sd adg05-01     cvol-01      adg05    0        524288   0        c1t1d4   ENA
pl cvol-02      cvol         ENABLED  ACTIVE   524288   CONCAT   -        RW
sd adg06-01     cvol-02      adg06    0        524288   0        c1t1d5   ENA
```

The result looks terrible, but don't give up! Some little drawing will do no harm:

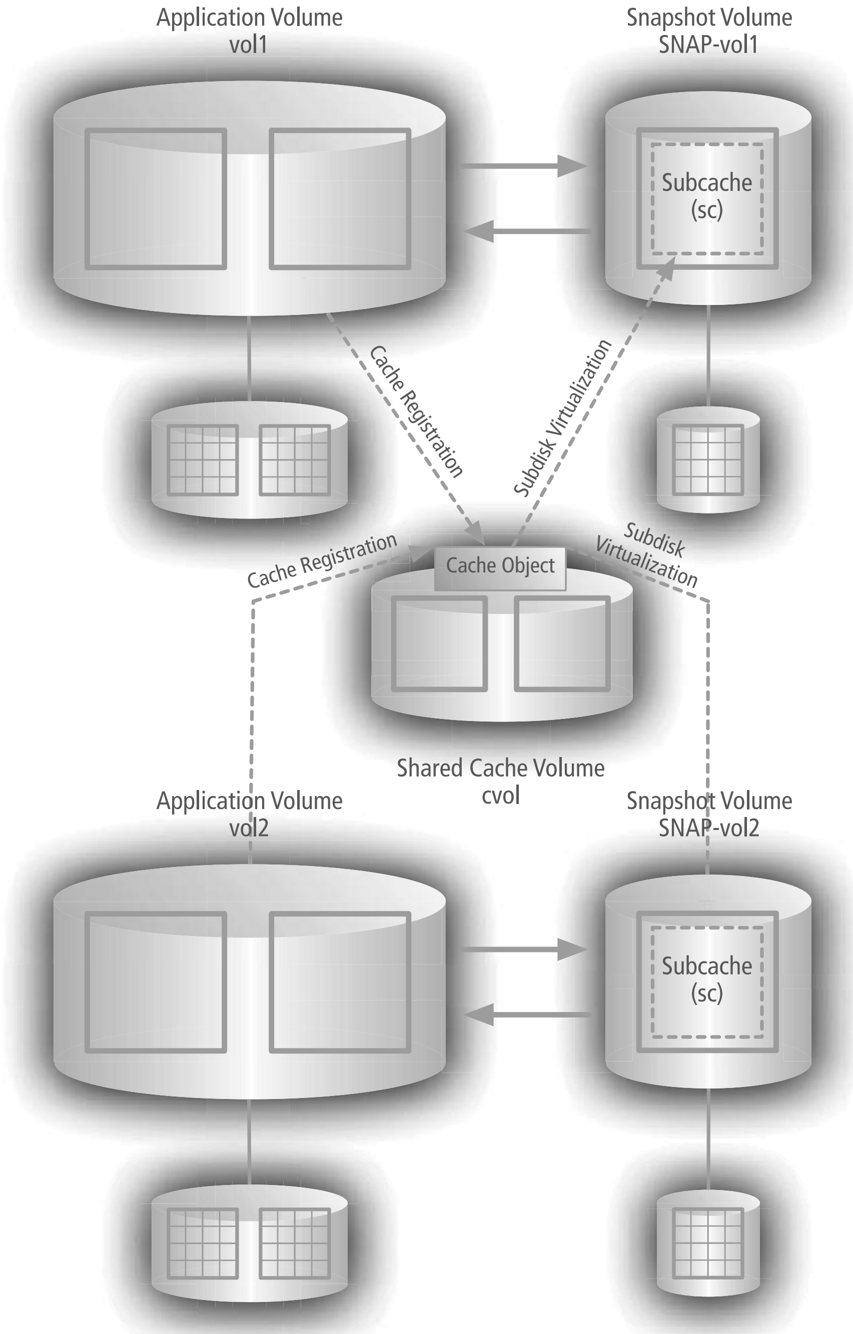


Figure 9-9: Space optimized snapshots of two volumes with shared cache

The picture together with some object names shows both application volumes (**vol1**, **vol2**, each mirrored) with the associated DCL volumes (**vol1_dcl**, **vol2_dcl**, also mirrored), the cache object **cobjcvol** with its mirrored cache volume **cvol** , both unmirrored snapshot volumes (**SNAP-vol1**, **SNAP-vol2**) with their associated DCL volumes (**SNAP-vol1_dcl**, **SNAP-vol2_dcl**), and, finally, the snap objects pointing from the application volumes to their snapshots and vice-versa. Remember: application volumes and snapshot volumes as well need DCL bitmap volumes to log write I/Os to them. After all, it is not so incomprehensible, as it looked at the first sight.

Let's write some data to the application volumes to test the snapshot mechanism (128 MB to **vol1**, 64 MB to **vol2**):

```
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol1 bs=1024k count=128
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol2 bs=1024k count=64
# vxsnap -g adg print
NAME    SNAPOBJECT    TYPE    PARENT    SNAPSHOT    %DIRTY    %VALID

vol1     --           volume   --       --           --       100.00
        SNAP-vol1_snp  volume   --       SNAP-vol1    12.50    --

vol2     --           volume   --       --           --       100.00
        SNAP-vol2_snp  volume   --       SNAP-vol2    6.25     --

SNAP-vol1 vol1_snp     volume   vol1     --           12.50    12.50

SNAP-vol2 vol2_snp     volume   vol2     --           6.25     6.25
```

Well, the output of the last command is disappointing to some extent. Indeed, compared to the size of the application volumes, we created an amount of 12.50% and 6.25% respectively of dirty regions. But we know, that 128 MB + 64 MB = 192 MB of data overwritten occupy already 75% of the cache volume (256 MB). Fortunately, VxVM provides a command to show the actual usage of the cache volume:

```
# vxcache stat cobjcvol
CACHE NAME                 TOTAL(Mb)    USED(Mb) (%)    AVAIL(Mb) (%)   SDCNT
cobjcvol                         256         196 (76)           60 (23)      2
```

Within the output, we recognize the name of the cache object, its total, used (4 MB in addition due to cache management data), and available size (note the slight rounding error in the percentage numbers), and the number of virtual snapshot subdisks simulated by the cache volume.

9.3.10 Autogrow Related Attributes

A detailed analysis of the cache object attributes reveals some further interesting features of the space optimized snapshot (excerpts):

```
# vxprint -g adg -m cobjcvol
cache cobjcvol
[…]
        autogrow=off
[…]
        hwmark=90
        autogrowby=104848
        max_autogrow=1048576
[…]
```

Currently, an attribute called **autogrow** seems to be turned off (you may specify **autogrow=on**, when creating the cache object). Another attribute called **hwmark** could mean a high water mark, obviously specified in percent unit. Reaching or exceeding the high water mark of the cache object could trigger an automated growth of its cache volume, probably by 104,848 sectors (about 51 MB, which is approximately 20% of the original cache volume size) defined by the attribute **autogrowby**. In case of subsequent cache limit events, the attribute **max_autogrow** seems to set a final limit to the cache volume size. Let's activate and test our assumptions by overwriting further 40 MB:

```
# vxcache -g adg set autogrow=on cobjcvol
# vxcache -g adg set max_autogrow=400m cobjcvol
# vxprint -g adg -F '%name %cachevol_len %autogrow %max_autogrow' cobjcvol
cobjcvol 524288 on 819200
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol2 bs=1024k count=40 oseek=64
```

After a few seconds, the cache volume has grown:

```
# vxcache -g adg stat cobjcvol
CACHE NAME                 TOTAL(Mb)    USED(Mb) (%)     AVAIL(Mb) (%)   SDCNT
cobjcvol                         307         236 (76)           71 (23)      2
# vxprint -rtg adg cvol
[…]
v  cvol         cobjcvol     ENABLED  ACTIVE   629136   SELECT    -        fsgen
pl cvol-01      cvol         ENABLED  ACTIVE   629136   CONCAT    -        RW
sd adg05-01     cvol-01      adg05    0        629136   0         c1t1d4   ENA
pl cvol-02      cvol         ENABLED  ACTIVE   629136   CONCAT    -        RW
sd adg06-01     cvol-02      adg06    0        629136   0         c1t1d5   ENA
```

Since the cache volume is mirrored, we expect, that VxVM issued a read-writeback synchronization thread for the additional volume size. We undertake to check for synchronization I/Os when triggering once again an autogrow of the cache volume:

```
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol2 bs=1024k count=64 oseek=104
# while :; do vxtask list | tail +2; done
…
 42015         RDWRBACK/R 50.79% 629136/733984/682384 RESYNC cvol adg
```

```
…
^C
# vxcache -g adg stat cobjcvol
CACHE NAME                 TOTAL(Mb)   USED(Mb) (%)    AVAIL(Mb) (%)   SDCNT
cobjcvol                         358        300 (83)          58 (16)      2
# vxprint -rtg adg cvol
[…]
v  cvol         cobjcvol     ENABLED  ACTIVE   733984   SELECT    -        fsgen
pl cvol-01      cvol         ENABLED  ACTIVE   733984   CONCAT    -        RW
sd adg05-01     cvol-01      adg05    0        733984   0         c1t1d4   ENA
pl cvol-02      cvol         ENABLED  ACTIVE   733984   CONCAT    -        RW
sd adg06-01     cvol-02      adg06    0        733984   0         c1t1d5   ENA
```

It worked once again! Furthermore, we could verify the predicted read-writeback synchronization. And finally, **root@localhost** already got two e-mails of the following sort:

```
[…]
Subject: Volume Manager cache grow notification on host haensel
[…]
Got a grow event notification for cache-volume cvol associated to cache-object
cobjcvol in disk-group adg
```

Another try! But remember: Our cache volume has a size of about 358 MB. The next autogrow event will try to add another 51 MB to it, which will exceed the defined maximum size of the cache volume (400 MB).

```
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol2 bs=1024k count=64 oseek=168
# vxcache -g adg stat cobjcvol
CACHE NAME                 TOTAL(Mb)   USED(Mb) (%)    AVAIL(Mb) (%)   SDCNT
cobjcvol                         358        236 (65)         122 (34)      1
```

Oops! No resize operation did happen! The amount of used cache volume space was even reduced! And, what is more, the number of virtual subdisks simulated by the cache object was decremented. This looks suspiciously like a damaged snapshot to **vol2**:

```
# vxprint -rLtg SNAP-vol2
[…]
v  SNAP-vol2    -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl SNAP-vol2-P01 SNAP-vol2   ENABLED  ACTIVE   2097152  CONCAT    -        RW
sc SNAP-vol2-S01 SNAP-vol2-P01 cobjcvol 0      2097152  0         -        ENA
dc SNAP-vol2_dco SNAP-vol2   SNAP-vol2_dcl

v  SNAP-vol2_dcl -           ENABLED  ACTIVE   544      SELECT    -        gen
pl SNAP-vol2_dcl-01 SNAP-vol2_dcl ENABLED ACTIVE 544    CONCAT    -        RW
sd adg08-01     SNAP-vol2_dcl-01 adg08 0       544      0         c1t1d7   ENA
sp vol2_snp     SNAP-vol2    SNAP-vol2_dco
```

No, this snapshot seems to work properly. What about **SNAP-vol1**?

```
# vxprint -rLtg adg SNAP-vol1
VxVM vxprint ERROR V-5-1-924 Record SNAP-vol1 not found
```

O my god! The "wrong" snapshot was destroyed! So, the first conclusion we draw from our observations, is to always set the **max_autogrow** attribute to an integer multiple of the **autogrowby** value plus the initial size of the cache volume. Another conclusion is not to rely too much on the autogrow features of the space optimized snapshots: A final cache overflow will destroy some of them. Furthermore, not tested by our investigations above, multiple fast I/Os writing on the application or snapshot volumes may be faster than the autogrow mechanism leading to destroyed or disabled snapshots. Consider this when defining the **hwmark** and **autogrowby** attribute values.

Note: VxVM 4.x did not destroy the snapshots, but disabled them. Nevertheless, the effect was the same: You could reuse the snapshots only by deleting and recreating them.

If you fear a soon cache overflow or your cache volume occupies too much storage, you may manually resize the cache volume. The **vxcache** command provides appropriate keywords for those operations: **growcacheby**, **growcacheto**, **shrinkcacheby**, and **shrinkcacheto**.

Who tells VxVM, that the cache volume has reached or exceeded the high water mark threshold? How is cache volume resizing performed? During the boot process, a script named **vxcached** is started into background, which itself invokes **vxnotify** with the option **-C** (cache events):

```
# ptree $(pgrep -xu0 vxcached)
2626  /sbin/sh - /usr/lib/vxvm/bin/vxcached root
  3583  /sbin/sh - /usr/lib/vxvm/bin/vxcached root
    3584  vxnotify -C -w 15
```

We recognize a process architecture similar to the **vxrelocd/vxsparecheck** and **vxconfigbackupd** processes explained in the troubleshooting chapter (see page 372). **vxnotify** is informed by the kernel about the cache event and generates standard output like the following:

```
grow on cachevolume cvol rid 0.8240 for cache cobjcvol rid 0.8254 dg adg dgid
1220261661.45.haensel
```

vxcached captures the output and invokes a command growing the cache volume by the defined amount of space, if possible. If necessary, you may create your own cache event handling with a self-written script comparable to **vxcached** - without Veritas support, of course.

9.3.11 Cascading Snapshots

The full sized instant and the space optimized volume snapshots provide another useful feature compared to the exclusively physical legacy full sized snapshot. Assume you want to create multiple snapshots on the same application device, e.g. hourly on a database volume each day for database recovery strategies against logical database errors such as inadvertently dropped tables. At 11:15 pm your database writes new data to a volume region still unchanged since mid-night. As you may remember, the legacy full-sized snapshot mechanism needs to copy the original data set to ALL existing snapshot volumes, before the new data set can be stored on the application volume. In our case, the database write I/O will have to wait for 24 (0:00 am to 11:00 pm) copy-on-first-writes - a performance drawback intolerable in enterprise environments!

In contrast, the instant snapshot mechanisms (full sized and space optimized) may maintain a cascading relationship by using the keyword **infrontof**: The original data set is copied only once to the latest snapshot device (full sized) or to the cache volume (space optimized), and the DCL volume bitmaps of the snapshots reflect the new location of these data - independent from the number of existing snapshots. We demonstrate the effect for a space optimized snapshot, but narrow a little bit the number of snapshots created at an hourly base:

```
# vxassist -g adg make vol 1g layout=mirror nmirror=2 init=active
# vxsnap -g adg prepare vol
# vxassist -g adg make cvol 256m layout=mirror nmirror=2 init=active
# vxmake -g adg cache cobjcvol cachevolname=cvol autogrow=on
# vxcache -g adg start cobjcvol
# vxsnap -g adg make source=vol/new=SP01-vol/cache=cobjcvol
# vxsnap -g adg make source=vol/new=SP02-vol/\
  infrontof=SP01-vol/cache=cobjcvol
# vxsnap -g adg make source=vol/new=SP03-vol/\
  infrontof=SP02-vol/cache=cobjcvol
# vxsnap -g adg make source=vol/new=SP04-vol/\
  infrontof=SP03-vol/cache=cobjcvol
# vxcache -g adg stat
CACHE NAME                  TOTAL(Mb)   USED(Mb) (%)    AVAIL(Mb) (%)   SDCNT
cobjcvol                          256          4 (1)          252 (98)      4
# vxsnap -g adg -n print
NAME         DG         OBJTYPE SNAPTYPE PARENT     PARENTDG   SNAPDATE
vol          adg        vol     -        -          -          - -
SP01-vol     adg        vol     spaceopt vol        adg        2008/09/21 09:00
SP02-vol     adg        vol     spaceopt vol        adg        2008/09/21 10:00
SP03-vol     adg        vol     spaceopt vol        adg        2008/09/21 11:00
SP04-vol     adg        vol     spaceopt vol        adg        2008/09/21 12:00
# vxprint -g adg -cF '%{assoc:-15} %creation_time' \
  -e 'sp_vol_name~/^SP0[1-4]-vol$/'
SP01-vol       Tue Sep 21 09:00:00 2008
SP02-vol       Tue Sep 21 10:00:00 2008
```

```
SP03-vol        Tue Sep 21 11:00:00 2008
SP04-vol        Tue Sep 21 12:00:00 2008
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol bs=1024k count=100
# vxcache -g adg stat
CACHE NAME                 TOTAL(Mb)    USED(Mb) (%)     AVAIL(Mb) (%)   SDCNT
cobjcvol                         256         104 (40)          152 (59)      4
```

9.3.12 A Final Example for Volume Snapshots

For all those of our readers still not satisfied by the complexity of snapshot structures, we provide the output of a **vxprint** command. Please decode! Note: This is a realistic scenario! If most of the data centers do not use advanced VxVM volume architectures, then it does probably not mean, that complex volumes are unnecessary, but that the administrators need (better) Storage Foundation courses.

```
v  SNAP-vol1    -            ENABLED  ACTIVE   419430400 SELECT   -        fsgen
pl SNAP-vol1-P01 SNAP-vol1   ENABLED  ACTIVE   419430400 CONCAT   -        RW
sc SNAP-vol1-S01 SNAP-vol1-P01 cobjcvol 0      419430400 0        -        ENA
dc SNAP-vol1_dco SNAP-vol1   SNAP-vol1_dcl

v  SNAP-vol1_dcl -            ENABLED  ACTIVE   7488     SELECT   -        gen
pl SNAP-vol1_dcl-01 SNAP-vol1_dcl ENABLED ACTIVE 7488   CONCAT    -        RW
sd adg05-01     SNAP-vol1_dcl-01 adg05 87607552 7488    0         c1t1d4   ENA
sp vol1_snp     SNAP-vol1    SNAP-vol1_dco


v  SNAP-vol2    -            ENABLED  ACTIVE   419430400 SELECT   -        fsgen
pl SNAP-vol2-P01 SNAP-vol2   ENABLED  ACTIVE   419430400 CONCAT   -        RW
sc SNAP-vol2-S01 SNAP-vol2-P01 cobjcvol 0      419430400 0        -        ENA
dc SNAP-vol2_dco SNAP-vol2   SNAP-vol2_dcl

v  SNAP-vol2_dcl -            ENABLED  ACTIVE   7488     SELECT   -        gen
pl SNAP-vol2_dcl-01 SNAP-vol2_dcl ENABLED ACTIVE 7488   CONCAT    -        RW
sd adg07-01     SNAP-vol2_dcl-01 adg07 87607552 7488    0         c1t1d6   ENA
sp vol2_snp     SNAP-vol2    SNAP-vol2_dco


v  vol1         -            ENABLED  ACTIVE   419430400 SELECT   vol1-03  fsgen
pl vol1-03      vol1         ENABLED  ACTIVE   419430400 STRIPE   2/512    RW

sv vol1-S01     vol1-03      vol1-L01 1        122107648 0/0      2/2      ENA
v2 vol1-L01     -            ENABLED  ACTIVE   122107648 SELECT   -        fsgen
p2 vol1-P01     vol1-L01     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg01-02     vol1-P01     adg01    0        122107648 0        c1t1d0   ENA
p2 vol1-P02     vol1-L01     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg03-02     vol1-P02     adg03    0        122107648 0        c1t1d2   ENA
```

```
sv vol1-S02     vol1-03      vol1-L02 1        87607552 0/122107648 2/2    ENA
v2 vol1-L02     -            ENABLED  ACTIVE   87607552 SELECT    -        fsgen
p2 vol1-P03     vol1-L02     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg05-02     vol1-P03     adg05    0        87607552 0         c1t1d4   ENA
p2 vol1-P04     vol1-L02     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg07-02     vol1-P04     adg07    0        87607552 0         c1t1d6   ENA

sv vol1-S03     vol1-03      vol1-L03 1        122107648 1/0      2/2      ENA
v2 vol1-L03     -            ENABLED  ACTIVE   122107648 SELECT   -        fsgen
p2 vol1-P05     vol1-L03     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg02-02     vol1-P05     adg02    0        122107648 0        c1t1d1   ENA
p2 vol1-P06     vol1-L03     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg04-02     vol1-P06     adg04    0        122107648 0        c1t1d3   ENA

sv vol1-S04     vol1-03      vol1-L04 1        87607552 1/122107648 2/2    ENA
v2 vol1-L04     -            ENABLED  ACTIVE   87607552 SELECT    -        fsgen
p2 vol1-P07     vol1-L04     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg06-02     vol1-P07     adg06    0        87607552 0         c1t1d5   ENA
p2 vol1-P08     vol1-L04     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg08-02     vol1-P08     adg08    0        87607552 0         c1t1d7   ENA
dc vol1_dco     vol1         vol1_dcl

v  vol1_dcl     -            ENABLED  ACTIVE   7488     SELECT    -        gen
pl vol1_dcl-01  vol1_dcl     ENABLED  ACTIVE   7488     CONCAT    -        RW
sd adg06-01     vol1_dcl-01  adg06    87607552 7488     0         c1t1d5   ENA
pl vol1_dcl-02  vol1_dcl     ENABLED  ACTIVE   7488     CONCAT    -        RW
sd adg08-01     vol1_dcl-02  adg08    87607552 7488     0         c1t1d7   ENA
sp SNAP-vol1_snp vol1        vol1_dco


v  vol2         -            ENABLED  ACTIVE   419430400 SELECT   vol2-03  fsgen
pl vol2-03      vol2         ENABLED  ACTIVE   419430400 STRIPE   2/512    RW

sv vol2-S01     vol2-03      vol2-L01 1        122107648 0/0      2/2      ENA
v2 vol2-L01     -            ENABLED  ACTIVE   122107648 SELECT   -        fsgen
p2 vol2-P01     vol2-L01     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg09-02     vol2-P01     adg09    0        122107648 0        c1t1d8   ENA
p2 vol2-P02     vol2-L01     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg11-02     vol2-P02     adg11    0        122107648 0        c1t1d10  ENA

sv vol2-S02     vol2-03      vol2-L02 1        87607552 0/122107648 2/2    ENA
v2 vol2-L02     -            ENABLED  ACTIVE   87607552 SELECT    -        fsgen
p2 vol2-P03     vol2-L02     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg13-02     vol2-P03     adg13    0        87607552 0         c1t1d12  ENA
p2 vol2-P04     vol2-L02     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg15-02     vol2-P04     adg15    0        87607552 0         c1t1d14  ENA
```

```
sv vol2-S03     vol2-03      vol2-L03 1        122107648 1/0      2/2      ENA
v2 vol2-L03     -            ENABLED  ACTIVE   122107648 SELECT   -        fsgen
p2 vol2-P05     vol2-L03     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg10-02     vol2-P05     adg10    0        122107648 0        c1t1d9   ENA
p2 vol2-P06     vol2-L03     ENABLED  ACTIVE   122107648 CONCAT   -        RW
s2 adg12-02     vol2-P06     adg12    0        122107648 0        c1t1d11  ENA

sv vol2-S04     vol2-03      vol2-L04 1        87607552 1/122107648 2/2    ENA
v2 vol2-L04     -            ENABLED  ACTIVE   87607552 SELECT    -        fsgen
p2 vol2-P07     vol2-L04     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg14-02     vol2-P07     adg14    0        87607552 0         c1t1d13  ENA
p2 vol2-P08     vol2-L04     ENABLED  ACTIVE   87607552 CONCAT    -        RW
s2 adg16-02     vol2-P08     adg16    0        87607552 0         c1t1d15  ENA
dc vol2_dco     vol2         vol2_dcl

v  vol2_dcl     -            ENABLED  ACTIVE   7488     SELECT    -        gen
pl vol2_dcl-01  vol2_dcl     ENABLED  ACTIVE   7488     CONCAT    -        RW
sd adg14-01     vol2_dcl-01  adg14    87607552 7488     0         c1t1d13  ENA
pl vol2_dcl-02  vol2_dcl     ENABLED  ACTIVE   7488     CONCAT    -        RW
sd adg16-01     vol2_dcl-02  adg16    87607552 7488     0         c1t1d15  ENA
sp SNAP-vol2_snp vol2        vol2_dco

co cobjcvol     cvol         ENABLED  ACTIVE

v  cvol         cobjcvol     ENABLED  ACTIVE   209715200 SELECT   cvol-03  fsgen
pl cvol-03      cvol         ENABLED  ACTIVE   209715200 STRIPE   2/128    RW

sv cvol-S01     cvol-03      cvol-L01 1        104857600 0/0      2/2      ENA
v2 cvol-L01     -            ENABLED  ACTIVE   104857600 SELECT   -        fsgen
p2 cvol-P01     cvol-L01     ENABLED  ACTIVE   104857600 CONCAT   -        RW
s2 adg17-02     cvol-P01     adg17    0        104857600 0        c1t1d16  ENA
p2 cvol-P02     cvol-L01     ENABLED  ACTIVE   104857600 CONCAT   -        RW
s2 adg19-02     cvol-P02     adg19    0        104857600 0        c1t1d18  ENA

sv cvol-S02     cvol-03      cvol-L02 1        104857600 1/0      2/2      ENA
v2 cvol-L02     -            ENABLED  ACTIVE   104857600 SELECT   -        fsgen
p2 cvol-P03     cvol-L02     ENABLED  ACTIVE   104857600 CONCAT   -        RW
s2 adg18-02     cvol-P03     adg18    0        104857600 0        c1t1d17  ENA
p2 cvol-P04     cvol-L02     ENABLED  ACTIVE   104857600 CONCAT   -        RW
s2 adg20-02     cvol-P04     adg20    0        104857600 0        c1t1d19  ENA
```

9.4 Veritas File System Based Snapshots

9.4.1 Cache Overflow on a Traditional Snapshot

The "Easy Sailing" section already described a snapshot mechanism based on VxFS, providing a completely logical snapshot with a mountable snapshot device storing only the originals of data overwritten by the application. We still didn't explain the snapshot behavior when exceeding the capacity of the snapshot device. Do we have something comparable to the autogrow feature of the volume based space optimized snapshot?

To get an answer, we will create a file system containing four files at 5 MB and an appropriate snapshot device (10% in size of the original file system). We choose the simplest volume layouts to indicate that we do not deal with volume based raw device snapshots and their plex break-off:

```
# vxassist -g adg make vol 100m
# mkfs -F vxfs /dev/vx/rdsk/adg/vol
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
# for i in 1 2 3 4; do mkfile 5m /mnt/file$i; done
# vxassist -g adg make snapvol 10m
# mount -F vxfs -o snapof=/mnt /dev/vx/dsk/adg/snapvol /mnt_snap
# ls -lA /mnt*
/mnt:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found

/mnt_snap:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   22645   74777    24%    /mnt
/dev/vx/dsk/adg/snapvol
                      102400   22645   74771    24%    /mnt_snap
```

Currently, the original and the snapshot file system contain exactly the same files, in other words, the bitmap of the snapshot device only points to the data set of the original file system. On first thought, we expect a cache overflow after removing two files from the original file system:

```
# rm /mnt/file1 /mnt/file2
# ls -lA /mnt*
/mnt:
total 20480
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found

/mnt_snap:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   12405   84377    13%    /mnt
/dev/vx/dsk/adg/snapvol
                      102400   22645   74771    24%    /mnt_snap
# od -cAd /mnt_snap/file1
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
5242880
# od -cAd /mnt_snap/file2
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
5242880
```

Hmm! Nothing happened except for a proper handling by the snapshot still providing the file contents of the files removed on the original file system (tested by an **od** read). We enter into an impatient testing instead of a calm deliberation:

```
# rm /mnt/file?
# ls -lA /mnt*
/mnt:
total 0
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found

/mnt_snap:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
```

```
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found
# df -k /mnt*
Filesystem           kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400    2165   93978     3%    /mnt
/dev/vx/dsk/adg/snapvol
                      102400   22645   74771    24%    /mnt_snap
# od -cAd /mnt_snap/file?
0000000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
20971520
```

What is this? Is it some kind of wizardry? No, because VxFS (as other file systems) did not completely remove the file. Some file system basics: How is a file stored within the file system? First, the directory file for the directory containing our file provides a mapping between the file name and the inode number, all this occupying only a few bytes. Since the directory file is updated by this operation, its inode reflects the new modification time, and potentially a new block is assigned for it. Secondly, an inode is allocated (in VxFS as part of an inode structural file, default size 256 bytes, configurable to 512 bytes) storing all the file attributes (such as file type, modification time, owner, permissions) and the required address-length-pairs (VxFS) to denote the physical location of the stored file contents. Thirdly and finally, we need storage for the data blocks ("extents" in VxFS) whose size summed up correspond to the file size (rounded up to the next block multiple).

What happens, if a file is removed from the file system? The entry mapping file name and inode number is cleared (VxFS) within the directory file, and the modification time of the directory file is updated. Furthermore, the inode, being now invalid, is removed from the inode structural file (VxFS). But the predominant amount of storage, the data blocks covering the file contents remain unchanged on the device as long as no new data are written to the file system. That's why our removal of all four files only copied a few metadata blocks to the snapshot device.

Furnished with the appropriate file system knowledge, we expect that overwriting one file on the original file system will not exceed the limit of the snapshot storage. But overwriting the second file will ... Well, be in for a surprise!

```
# mkfile 5m /mnt/file1
# ls -lA /mnt*
/mnt:
total 10254
-rw------T   1 root     root     5242880 Sep 21 09:18 file1
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found

/mnt_snap:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
```

```
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400    7285   89178     8%    /mnt
/dev/vx/dsk/adg/snapvol
                      102400   22645   74771    24%    /mnt_snap
# mkfile 5m /mnt/file2
# ls -lA /mnt*
/mnt:
total 20480
-rw------T   1 root     root     5242880 Sep 21 09:18 file1
-rw------T   1 root     root     5242880 Sep 21 09:20 file2
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found

/mnt_snap:
total 40960
-rw------T   1 root     root     5242880 Sep 21 08:19 file1
-rw------T   1 root     root     5242880 Sep 21 08:19 file2
-rw------T   1 root     root     5242880 Sep 21 08:19 file3
-rw------T   1 root     root     5242880 Sep 21 08:19 file4
drwxr-xr-x   2 root     root          96 Sep 21 08:18 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   12405   84377    13%    /mnt
/dev/vx/dsk/adg/snapvol
                      102400   22645   74771    24%    /mnt_snap
```

Did we unfoundedly gloat with our file system knowledge? No, the piece of information provided by the command output was stored still within the file system kernel cache. But the system console shows a warning message, and an **od** command (or other read accesses) is unable to read the file content (which produces another console message):

```
Sep 21 09:20:12 haensel vxfs: WARNING: msgcnt 1 mesg 028: V-2-28: vx_snap_alloc
- /dev/vx/dsk/adg/snapvol snapshot file system out of space

# od -cAd /mnt_snap/file2
0000000

Sep 21 09:20:55 haensel vxfs: WARNING: msgcnt 2 mesg 032: V-2-32: vx_disable - /
dev/vx/dsk/adg/snapvol snapshot file system disabled
```

To sum up: A physical overflow of the snapshot device will disable the snapshot file system making all snapped data inaccessible. There is no way to recover the snapshot, and, what is more, there is no way to show the current quota of the snapshot device while the snapshot is still enabled. So, it is a good idea to choose a somewhat oversized snapshot device and not to rely for too long a period on its proper functioning. But don't forget the main advantage of this kind of snapshot: It is cheap in storage and license costs.

Two final remarks: A file system based access to the snapshot via its mount point (such as **ls**, **find**, **tar**) does not show any particularity, the snapshot behaves like a regular mounted file system. A standard raw device access to the snapshot storage (e.g. by **dd**) only gets the physical snapshot storage device data, because the pointing bitmap is not understood. If you want to perform a valid backup of your snapshot file system close to raw device access, you must use the **vxdump** tool (and for restore purposes the corresponding **vxrestore** command).

Refreshing a VxFS snapshot (even a disabled one) is quite easy: Just unmount the snapshot and mount it once again. You may restore the file system content of the application file system by simply copying the required files from the snapshot mount to the application mount. If the amount of copied files will exceed the capacity of the snapshot device (the snapshot will handle those copy operations as overwritten or new files on its original file system), you must copy your files to a temporary staging file system. To delete a snapshot: Unmount it and remove the snapshot device.

9.4.2 VxFS Storage Checkpoints

General Concept

Let's turn to a snapshot concept really deserving to be called an intelligent mechanism suitable for enterprise needs! To understand its capabilities and advantages (and only a few weaknesses), we recall the flexible layout of VxFS (in the following example on a 128 MB volume with a 10 MB file on VxFS version 7 layout), as shown by the **ncheck** command:

```
# ncheck -F vxfs -o sector= /dev/vx/rdsk/adg/vol
/dev/vx/rdsk/adg/vol:

sectors(204800)        blocks(0)
-----------------      -----------------
0/0-0/204799           0/0-0/102400

fileset    fset        match  match devid/
name       indx  inode indx   inode sectors       name
---------- ---- ------ ---- ------ ------------ -----------------
STRUCTURAL    1      3    -     35 0/18-0/21     <fileset_header>
STRUCTURAL    1      4    1      - 0/22-0/29     <inode_alloc_unit>
STRUCTURAL    1      5    1     37 0/4640-0/4655 <inode_list>
STRUCTURAL    1      5    1     37 0/48-0/63     <inode_list>
STRUCTURAL    1      5    1     37 0/4624-0/4639 <inode_list>
STRUCTURAL    1      5    1     37 0/32-0/47     <inode_list>
STRUCTURAL    1      6    -      - 0/30-0/31     <current_usage_tbl>
STRUCTURAL    1      7    -     39 0/64-0/65     <object_loc_tbl>
STRUCTURAL    1      8    -     40 0/80-0/1103   <device_config>
STRUCTURAL    1      9    -     41 0/1104-0/3151 <intent_log>
STRUCTURAL    1     11    -      - 0/66-0/67     <fs_allocation_policy>
STRUCTURAL    1     32    -      - 0/68-0/69     <history_log>
STRUCTURAL    1     33    -      - 0/4614-0/4615 <device_label>
STRUCTURAL    1     33    -      - 0/0-0/17      <device_label>
STRUCTURAL    1     35    -      3 0/4608-0/4611 <fileset_header>
STRUCTURAL    1     37    1      5 0/4640-0/4655 <inode_list>
STRUCTURAL    1     37    1      5 0/48-0/63     <inode_list>
STRUCTURAL    1     37    1      5 0/4624-0/4639 <inode_list>
STRUCTURAL    1     37    1      5 0/32-0/47     <inode_list>
STRUCTURAL    1     39    -      7 0/4612-0/4613 <object_loc_tbl>
STRUCTURAL    1     40    -      8 0/4656-0/5679 <device_config>
STRUCTURAL    1     41    -      9 0/1104-0/3151 <intent_log>
STRUCTURAL    1     64  999      - 0/70-0/77     <inode_alloc_unit>
STRUCTURAL    1     65  999     97 0/3152-0/3167 <inode_list>
STRUCTURAL    1     69  999      - 0/3168-0/3183 <bsd_quota>
STRUCTURAL    1     70  999      - 0/3184-0/3199 <bsd_quota>
STRUCTURAL    1     71    -      - 0/78-0/79     <state_alloc_bitmap>
STRUCTURAL    1     72    -      - 0/3200-0/3201 <extent_au_summary>
STRUCTURAL    1     73    -    105 0/3264-0/3279 <extent_map>
STRUCTURAL    1     73    -    105 0/3232-0/3263 <extent_map>
STRUCTURAL    1     73    -    105 0/3216-0/3231 <extent_map>
STRUCTURAL    1     97  999     65 0/3152-0/3167 <inode_list>
STRUCTURAL    1    105    -     73 0/3264-0/3279 <extent_map>
STRUCTURAL    1    105    -     73 0/3232-0/3263 <extent_map>
STRUCTURAL    1    105    -     73 0/3216-0/3231 <extent_map>
UNNAMED     999      4    -      - 0/16384-0/36863 /file.10m
-             -      -    -      - 0/3202-0/3215 <free>
-             -      -    -      - 0/3280-0/4607 <free>
-             -      -    -      - 0/4616-0/4623 <free>
-             -      -    -      - 0/5680-0/16383 <free>
-             -      -    -      - 0/36864-0/204799 <free>
```

The first column reveals two file system instances to the raw device. First, the **STRUCTURAL** file set carrying index 1 (column 2) accesses the "files" storing general file system metadata such as the intent log, two object location tables to store the current position of most metadata files, extent maps, and so on (last column; for redundancy purposes, those metadata are addressed mostly by two inodes, see column 3 and 5). Furthermore, the **STRUCTURAL** file set contains metadata for its own file set (match index 1 in column 4) and for the current file system as visible by the virtual file system of the operating system and in use by an application (match index 999). Secondly, we recognize the standard file system for application purposes, called **UNNAMED** and carrying file set index 999, and, in our example, the physical location of a 10 MB file (column 6; 0 before the slash denotes the volume counter within a volume set, the numbers after the slash indicate start and end sectors). Free space on the file system device is listed at the end.

Please recall the required procedure of other snapshot mechanisms when overwriting an existing file or data set: The original file or data set must be copied to a snapshot container outside of the application device (snapshot device, cache device), before the new file or data set can be written to the application device. We have a noticeable performance drawback by additional I/Os.

The VxFS snapshot mechanism called "Storage Checkpoint" does not need a separate snapshot container, because it uses free space within the same device for snapshot purposes. To distinguish between the active file system and a snapshot file system, VxFS simply adds another file system instance to the device (besides **STRUCTURAL** and **UNNAMED**) arbitrarily named (we will choose "**CP**" and a time stamp) and with a partially own set of metadata. As long as the file system remains unchanged, both file system instances' metadata point to the same file contents.

```
# fsckptadm create CP$(date +%H%M) /mnt
# mount -F vxfs -o remount /dev/vx/dsk/adg/vol /mnt
# ncheck -F vxfs -o sector= /dev/vx/rdsk/adg/vol
/dev/vx/rdsk/adg/vol:

sectors(204800)         blocks(0)
-----------------       -----------------
0/0-0/204799            0/0-0/102400

fileset    fset        match  match devid/
name       indx  inode indx  inode sectors       name
---------- ---- ------ ---- ------ ------------- ------------------
STRUCTURAL    1      3    -     35 0/3280-0/3295 <fileset_header>
STRUCTURAL    1      3    -     35 0/18-0/21     <fileset_header>
STRUCTURAL    1      4    1      - 0/22-0/29     <inode_alloc_unit>
STRUCTURAL    1      5    1     37 0/4640-0/4655 <inode_list>
STRUCTURAL    1      5    1     37 0/48-0/63     <inode_list>
STRUCTURAL    1      5    1     37 0/4624-0/4639 <inode_list>
STRUCTURAL    1      5    1     37 0/32-0/47     <inode_list>
STRUCTURAL    1      6    -      - 0/30-0/31     <current_usage_tbl>
STRUCTURAL    1      7    -     39 0/64-0/65     <object_loc_tbl>
STRUCTURAL    1      8    -     40 0/80-0/1103   <device_config>
STRUCTURAL    1      9    -     41 0/1104-0/3151 <intent_log>
STRUCTURAL    1     11    -      - 0/66-0/67     <fs_allocation_policy>
STRUCTURAL    1     32    -      - 0/68-0/69     <history_log>
STRUCTURAL    1     33    -      - 0/4614-0/4615 <device_label>
STRUCTURAL    1     33    -      - 0/0-0/17      <device_label>
STRUCTURAL    1     35    -      3 0/196608-0/196623 <fileset_header>
STRUCTURAL    1     35    -      3 0/4608-0/4611 <fileset_header>
STRUCTURAL    1     37    1      5 0/4640-0/4655 <inode_list>
STRUCTURAL    1     37    1      5 0/48-0/63     <inode_list>
STRUCTURAL    1     37    1      5 0/4624-0/4639 <inode_list>
STRUCTURAL    1     37    1      5 0/32-0/47     <inode_list>
STRUCTURAL    1     39    -      7 0/4612-0/4613 <object_loc_tbl>
STRUCTURAL    1     40    -      8 0/4656-0/5679 <device_config>
STRUCTURAL    1     41    -      9 0/1104-0/3151 <intent_log>
STRUCTURAL    1     64  999      - 0/70-0/77     <inode_alloc_unit>
STRUCTURAL    1     65  999     97 0/3152-0/3167 <inode_list>
STRUCTURAL    1     69  999      - 0/3168-0/3183 <bsd_quota>
STRUCTURAL    1     70  999      - 0/3184-0/3199 <bsd_quota>
STRUCTURAL    1     71    -      - 0/78-0/79     <state_alloc_bitmap>
STRUCTURAL    1     72    -      - 0/3200-0/3201 <extent_au_summary>
STRUCTURAL    1     73    -    105 0/3264-0/3279 <extent_map>
STRUCTURAL    1     73    -    105 0/3232-0/3263 <extent_map>
STRUCTURAL    1     73    -    105 0/3216-0/3231 <extent_map>
STRUCTURAL    1     74 1000      - 0/3208-0/3215 <inode_alloc_unit>
STRUCTURAL    1     75 1000     76 0/5680-0/5695 <inode_list>
STRUCTURAL    1     76 1000     75 0/5680-0/5695 <inode_list>
STRUCTURAL    1     81 1000      - 0/3296-0/3311 <bsd_quota>
STRUCTURAL    1     82 1000      - 0/3312-0/3327 <bsd_quota>
STRUCTURAL    1     97  999     65 0/3152-0/3167 <inode_list>
STRUCTURAL    1    105    -     73 0/3264-0/3279 <extent_map>
STRUCTURAL    1    105    -     73 0/3232-0/3263 <extent_map>
STRUCTURAL    1    105    -     73 0/3216-0/3231 <extent_map>
UNNAMED     999      4    -      - 0/16384-0/36863 /file
-             -      -    -      - 0/3202-0/3207 <free>
-             -      -    -      - 0/3328-0/4607 <free>
-             -      -    -      - 0/4616-0/4623 <free>
-             -      -    -      - 0/5696-0/16383 <free>
-             -      -    -      - 0/36864-0/196607 <free>
-             -      -    -      - 0/196624-0/204799 <free>
```

Note the new file set match index 1000 in the fourth column of the **ncheck** output providing a separate inode allocation unit, an inode list file addressed by two inodes, and two BSD quota files. The first column does not list the new checkpoint, as it still does not differ from the active file system. The command **fsckptadm** to create the snapshot will be explained in more detail, of course. Please be aware, that **ncheck** operates on the raw device, while **fsckptadm** defines the storage checkpoint based on the mount point, i.e. by using the block device driver. In order to immediately demonstrate the effect of file system modifications by **ncheck**, the file system caches need to be flushed to the raw device, which is best performed by a remount (keeps read caches valid by flushing all dirty blocks).

Since we did not create another snapshot device, we must use the application block device driver to mount the storage checkpoint by specifying the storage checkpoint instance of the file system. But unlike the legacy VxFS snapshot, the time the checkpoint was created defines its time stamp, not the time it was mounted. The storage checkpoint may not be mounted to work as a snapshot.

```
# fsckptadm list /mnt
/mnt
CP1203:
        ctime                =  Wed Sep 21 12:03:11 2008
        mtime                =  Wed Sep 21 12:03:11 2008
        flags                =  largefiles
# mkdir /mnt_CP1203
# mount -F vxfs -o ckpt=CP1203 /dev/vx/dsk/adg/vol:CP1203 /mnt_CP1203
# ls -lA /mnt*
/mnt:
total 20480
-rw-------   1 root     root     10485760 Sep 21 10:19 file
drwxr-xr-x   2 root     root          96 Sep 21 10:17 lost+found

/mnt_CP1203:
total 0
-rw-------   1 root     root     10485760 Sep 21 10:19 file
drwxr-xr-x   2 root     root          96 Sep 21 10:17 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   12449   84336    13%    /mnt
/dev/vx/dsk/adg/vol:CP1203
                      102400   12449   84336    13%    /mnt_CP1203
```

A snapshot is intended to provide a frozen image in spite of write I/Os. Let's overwrite our file! We do not use the **mkfile** command immediately on our file system, because it shows some strange behavior when applied to a VxFS file system (zero device space reserved, but not actually written).

```
# mkfile 10m /tmp/file
# cp /tmp/file /mnt
# mount -F vxfs -o remount /dev/vx/dsk/adg/vol /mnt
# ncheck -F vxfs -o sector= /dev/vx/rdsk/adg/vol
...
UNNAMED     999      4    -      - 0/36864-0/40959 /file
UNNAMED     999      4    -      - 0/49152-0/65535 /file
CP1203     1000      4    -      - 0/16384-0/36863 /file
...
# ls -lA /mnt*
/mnt:
total 20480
-rw-------   1 root     root     10485760 Sep 21 12:55 file
drwxr-xr-x   2 root     root          96 Sep 21 10:17 lost+found

/mnt_CP1203:
total 20480
-rw-------   1 root     root     10485760 Sep 21 10:19 file
drwxr-xr-x   2 root     root          96 Sep 21 10:17 lost+found
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   22689   74736    24%    /mnt
/dev/vx/dsk/adg/vol:CP1203
                      102400   22689   74736    24%    /mnt_CP1203
```

The storage used by the original file (sectors 16384-36863, assigned to **UNNAMED**, see the previous **ncheck** output) is now assigned to the storage checkpoint **CP1203** only, i.e. the metadata set of **CP1203** keeps its information on that file showing its former content and attributes. The "overwriting" new file got a previously free storage location (in our case fragmented into two pieces due to the internal extent organization of VxFS) with own attributes (visible at the different modification time stamp in the **ls** output).

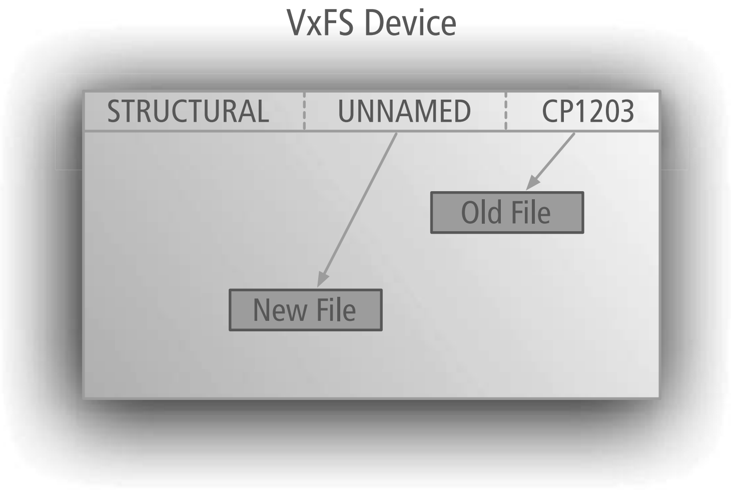


Figure 9-10: No "Copy on First Write" using VxFS Storage Checkpoints

Unlike the snapshot mechanisms we hitherto described in this chapter, the VxFS storage checkpoint does NOT perform a copy-on-first-write I/O! The application does NOT suffer from remarkable performance drawback! We will demonstrate some reasonable exceptions from that general rule in the "Technical Deep Dive" section. But in spite of some official Veritas/Symantec documentation, a copy-on-first-write is not the general I/O rule (one of the extremely rare situations a company sells its products with deteriorating information).

Storage Checkpoint Administration

Our example above already used the **fsckptadm** command to create a storage checkpoint. To see some statistical details of the active file system and the storage checkpoint, you may add the option **-v** ("verbose"):

```
# fsckptadm -v create CP$(date +%H%M) /mnt
CP1503:
        ctime                =  Wed Sep 21 15:03:26 2008
        mtime                =  Wed Sep 21 15:03:26 2008
        flags                =  largefiles
        # of inodes          =  32
        # of blocks         =  0
        # of reads          =  0
        # of writes         =  0
        # of pushes         =  0
        # of pulls          =  0
        # of moves          =  0
        # of move alls      =  0
        # of merge alls     =  0
        # of logged pushes  =  0
        # of enospc retries =  0
        # of overlay bmaps  =  0
```

Unfortunately, the meaning of the output is not officially reported, and only few entries are self-explaining. **ctime** denotes the creation time of the checkpoint, **mtime** the "modification" time stamp of the last write access to the checkpoint (if mounted read-write). Some file system or checkpoint attributes are listed under **flags**. The counters to **inodes**, (data) **blocks**, (file) **read** and **write** accesses could give some useful hints on file system usage, but are, needless to say, zeroed at checkpoint creation time. The same output may be produced at a later stage, after some overwritten files or blocks on the **UNNAMED** file system instance or some direct reads and writes to the checkpoint instance:

```
# fsckptadm -v list /mnt
/mnt
CP1203:
        ctime               =  Wed Sep 21 12:03:11 2008
        mtime               =  Wed Sep 21 15:54:26 2008
        flags               =  largefiles
        # of inodes         =  32
        # of blocks         =  10240
        # of reads          =  2
        # of writes         =  1
        # of pushes         =  0
        # of pulls          =  0
        # of moves          =  0
        # of move alls      =  0
        # of merge alls     =  0
        # of logged pushes  =  0
        # of enospc retries =  0
        # of overlay bmaps  =  0
```

Adding the option **-l** to the last mentioned command includes statistics to the **UNNAMED** file system instance. Omitting all options provides a short overview of existing storage checkpoints together with time stamps and flags (use **fsckptadm info** to display only one checkpoint):

```
# fsckptadm list /mnt
/mnt
CP1503:
        ctime                = Wed Sep 21 15:03:26 2008
        mtime                = Wed Sep 21 15:03:26 2008
        flags                = largefiles
CP1203:
        ctime                = Wed Sep 21 12:03:11 2008
        mtime                = Wed Sep 21 15:54:26 2008
        flags                = largefiles, mounted
```

If the cache volume of a space optimized volume snapshot gets out of space (**autogrow** disabled or maximum size for autogrow reached), snapshot volumes become disabled or are completely deleted. If copy-on-first-write operations overflow the cache device for the legacy VxFS snapshot, the snapshot will be disabled. In both cases, the mentioned snapshot behavior is not configurable. VxFS storage checkpoints provide a configurable flag called **removable**. If the file system device holding the active file system and the storage checkpoints as well runs out of space, you may decide what should happen: Should a checkpoint be removed to free space in favor of the running application (flag **removable** set), or should the checkpoint be kept, while application write I/Os are prohibited (**removable** cleared)? You may specify a removable checkpoint by adding the option **-r** when creating it. But at any time you may toggle the **removable** flag value by issuing a **fsckptadm set|clear** command, as shown in the following, somewhat disappointing example (file system 100 MB in size):

```
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
# mkfile 20m /tmp/file
# cp /tmp/file /mnt/file0
# cp /tmp/file /mnt/file1
# cp /tmp/file /mnt/file2
# fsckptadm create Ckpt /mnt
# mkdir /mnt_ckpt
# mount -F vxfs -o ckpt=Ckpt /dev/vx/dsk/adg/vol:Ckpt /mnt_ckpt
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   63649   36336    64%    /mnt
/dev/vx/dsk/adg/vol:Ckpt
                      102400   63649   36336    64%    /mnt_ckpt
# cp /tmp/file /mnt/file0
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   84129   17136    84%    /mnt
/dev/vx/dsk/adg/vol:Ckpt
                      102400   84129   17136    84%    /mnt_ckpt
# cp /tmp/file /mnt/file1
cp: /mnt/file1: No space left on device
# fsckptadm list /mnt
/mnt
Ckpt:
        ctime                = Thu Sep 21 08:44:56 2008
        mtime               =  Thu Sep 21 08:44:56 2008
        flags               =  largefiles, mounted
# fsckptadm set remove Ckpt /mnt
# fsckptadm list /mnt
/mnt
Ckpt:
        ctime               =  Thu Sep 21 08:44:56 2008
        mtime               =  Thu Sep 21 08:44:56 2008
        flags               =  largefiles, removable, mounted
# cp /tmp/file /mnt/file1
cp: /mnt/file1: No space left on device
```

Rats! Why is the checkpoint not removed in favor of the running application? Well, the checkpoint is still in use, because it is mounted. Our hope is, that we only need to unmount it in order to make it actually removable. Next try:

```
# umount /mnt_ckpt
# cp /tmp/file /mnt/file1
cp: /mnt/file1: No space left on device
```

Wow! That looks bad! We consult the manual page to **fsckptadm** and note an imprecise expression:

```
Under some conditions, when the file system runs out of space, removable Storage
Checkpoints are deleted.
```

Consulting the VxFS Administrator's Guide with its vague allusions to that topic, we get the impression, that database I/Os keeping the preallocated space for the database file at the same position by overwriting only some blocks within the file will produce an **ENOSPC** event ("Error: No space"). Let's start once again at the very beginning with database like I/O using Perl (the Shell is unable to write into an existing file without changing the file size):

```
# umount /mnt
# mkfs -F vxfs /dev/vx/rdsk/adg/vol
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
# mkfile 80m /tmp/file
# cp /tmp/file /mnt
# df -k /mnt
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   84085   17178    84%    /mnt
# fsckptadm -r create Ckpt /mnt
# fsckptadm list /mnt
/mnt
Ckpt:
        ctime               =  Thu Sep 21 09:23:21 2008
        mtime               =  Thu Sep 21 09:23:21 2008
        flags                =  largefiles, removable
```

The following Perl statement will overwrite a region of 10 MB at the beginning of the database file (for details see the comments at the end of each line). Since our file system device still holds about 17 MB free space, we do not expect a removal of the snapshot.

```
# perl -e '
  $m10=1024*1024*10;               # define 10 MB
  $Block="x" x $m10;               # a block 10 MB in size
  open(FH,"+< /mnt/file") || die;  # open read-write access by keeping the file
  sysseek(FH,0,0);                 # set file pointer to beginning of file
  syswrite(FH,$Block,$m10,0);      # write 10 MB block
  close(FH);'                      # close file
# df -k /mnt
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   94377    7529    93%    /mnt
# ls -l /mnt
total 163840
-rw-------   1 root     root     83886080 Sep 21 09:36 file
drwxr-xr-x   2 root     root          96 Sep 21 09:17 lost+found
# fsckptadm list /mnt
/mnt
Ckpt:
        ctime                =  Thu Sep 21 09:23:21 2008
        mtime                =  Thu Sep 21 09:23:21 2008
        flags                =  largefiles, removable
```

Correct, the file system usage increased by approximately 10 MB. Now the final blow! The next 10 MB region will be overwritten, thus blasting the space still available within the file system.

```
# perl -e '
  $m10=1024*1024*10;
  $Block="x" x $m10;
  open(FH,"+< /mnt/file") || die;
  sysseek(FH,$m10,0);              # set file pointer to 10 MB offset
  syswrite(FH,$Block,$m10,0);
  close(FH);'
# df -k /mnt
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol   102400   84101   17163    84%    /mnt
# ls -l /mnt
total 163840
-rw-------   1 root     root     83886080 Sep 21 09:40 file
drwxr-xr-x   2 root     root          96 Sep 21 09:17 lost+found
# fsckptadm list /mnt
/mnt
```

```
# fsckptadm -lv list /mnt
/mnt
UNNAMED:
        ctime                = Thu Sep 21 09:17:57 2008
        mtime                = Thu Sep 21 09:19:02 2008
        flags                = largefiles, mounted,
        # of inodes          = 32
        # of blocks          = 84085
        # of reads           = 0
        # of writes          = 15
        # of pushes          = 292
        # of pulls           = 0
        # of moves           = 0
        # of move alls       = 0
        # of merge alls      = 0
        # of logged pushes   = 1
        # of enospc retries  = 1
        # of overlay bmaps   = 0
```

Finally we got it! We already noticed that the Perl script needed more time to execute compared to the previous one due to checkpoint deletion. The file system space held by the storage checkpoint was freed, the checkpoint removed, and the detailed output to the active file system indeed displays an **enospc** ("Error: No space") event evoking the checkpoint removal.

The sequence of removal in case of multiple checkpoints is determined by their age ("first in, first out") and the priority of nodata storage checkpoints (see next paragraph) over data checkpoints.

Further flags of storage checkpoints are of less interest, so we will refer to them only in few words. A checkpoint may be marked as non-mountable (flag **nomount**), either by creating it (option **-u**) or by setting it afterwards (**fsckptadm set nomount** ...), thus prohibiting undesired access by non-root users (a root user may always clear the **nomount** flag).

A **nodata** checkpoint provides a snapshot for the file system metadata (such as file attributes, extent addresses), but not for file contents (option **-n** when creating, **fsckptadm set nodata** ... later). Issuing an **ncheck** command reveals, that only the metadata set is created for the snapshot even by modified file contents. The snapped file system metadata may serve, to mention an example, as a source to decide which files to save by an incremental backup. Naturally, a nodata storage checkpoint can never be converted to a data checkpoint.

As an intelligent snapshot mechanism, multiple VxFS storage checkpoints do not produce an overwhelming amount of additional I/Os. If a file is completely replaced by a new version, only the UNNAMED file system instance redirects its pointer to the new file version, while all existing checkpoints simply remain unchanged. If a VxFS checkpointed file system indeed performs a copy-on-first-write (e.g. by a database I/O, see the "Technical Deep Dive" section), the current file system instance keeps its file data block addresses, and all checkpoints redirect their addresses to the saved file block.

You cannot refresh a storage checkpoint to the current data set of the active file system by a single step. Instead, create a new storage checkpoint and, if you need to free space on the device or do not want to keep previous storage checkpoints, simply remove them.

Recovering a file system by a storage checkpoint may accomplished by three ways:

1. Mount the appropriate checkpoint to a temporary mount point (if it is not already mounted) and copy only those files to the application mount point you want to recover. A complete file system recovery using this procedure is space consuming, because the current files of the application mount are kept within the device for the still existing checkpoints. Destroy all unnecessary checkpoints to get more space.
2. Unmount the regular file system instance for the application (probably UNNAMED) and mount the desired checkpoint at the application mount point. Then restart (and recover) your application. All checkpoints created after the currently mounted one lose their snapshot functionality. Therefore, they should be removed. Unfortunately the checkpoint based file system instance remains a checkpoint, so we need to mount it by specifying its checkpoint name (and accordingly to modify entries in **/etc/vfstab** or in cluster resource configurations, and so on).
3. Therefore, a complete file system recovery by reactivating a storage checkpoint should also link the default file system instance to the checkpoint, not to UNNAMED anymore. VxFS provides an executable to do so:

```
# fsckptadm list /mnt
/mnt
CP1200:
        ctime                =  Sun Sep 28 12:00:00 2008
        mtime                =  Sun Sep 28 12:00:00 2008
        flags                =  largefiles
CP1100:
        ctime                =  Sun Sep 28 11:00:00 2008
        mtime                =  Sun Sep 28 11:00:00 2008
        flags                =  largefiles
CP1000:
        ctime                =  Sun Sep 28 10:00:00 2008
        mtime                =  Sun Sep 28 10:00:00 2008
        flags                =  largefiles
CP0900:
        ctime                =  Sun Sep 28 09:00:00 2008
        mtime                =  Sun Sep 28 09:00:00 2008
        flags                =  largefiles
# umount -f /mnt
# fsckpt_restore -l /dev/vx/dsk/adg/vol
/dev/vx/dsk/adg/vol:

UNNAMED:
        ctime                =  Sun Sep 28 08:53:10 2008
        mtime               =  Sun Sep 28 08:53:10 2008
        flags               =  largefiles, file system root

CP1200:
        ctime               =  Sun Sep 28 12:00:00 2008
        mtime               =  Sun Sep 28 12:00:00 2008
        flags               =  largefiles

CP1100:
        ctime               =  Sun Sep 28 11:00:00 2008
        mtime               =  Sun Sep 28 11:00:00 2008
        flags               =  largefiles

CP1000:
        ctime               =  Sun Sep 28 10:00:00 2008
        mtime               =  Sun Sep 28 10:00:00 2008
        flags               =  largefiles

CP0900:
        ctime               =  Sun Sep 28 09:00:00 2008
        mtime               =  Sun Sep 28 09:00:00 2008
        flags               =  largefiles


Select storage checkpoint for restore operation
 or <Control/D> (EOF) to exit
 or <Return> to list storage checkpoints: CP1000
CP1000:
        ctime               =  Sun Sep 28 10:00:00 2008
        mtime               =  Sun Sep 28 10:00:00 2008
        flags               =  largefiles

UX:vxfs fsckpt_restore: WARNING: V-3-24640: Any file system changes or storage
checkpoints
made after Sun Sep 28 10:00:00 2008 will be lost.

Restore the file system from storage checkpoint CP1000 ? (ynq) y
(Yes)
UX:vxfs fsckpt_restore: INFO: V-3-23760: File system restored from CP1000
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
# fsckptadm -l list /mnt
/mnt
CP1000:
        ctime               =  Sun Sep 28 10:00:00 2008
        mtime               =  Sun Sep 28 12:36:19 2008
        flags               =  largefiles, mounted,
CP0900:
        ctime                =  Sun Sep 28 09:00:00 2008
        mtime                =  Sun Sep 28 09:00:00 2008
        flags                =  largefiles
```

9.5 Creating a Full Sized Volume Snapshot Using Low-Level Commands

In order to create a snapshot without data change object (DCO) and data change log volume (DCL) we may issue some basic VxVM commands. This kind of snapshot, however, does not provide several advanced features of the standard snapshots created by the **vxsnap** command: no fast mirror resynchronization, no instant availability or instant refresh of the snapshot, and no space optimizing strategies. Our basic snapshot procedure is a simple plex break-off and, when reattaching it to its original volume, a simple plex attach operation. Our example will be supplemented with some vxstat commands to verify procedure and amount of synchronization.

Step 1 We create the application volume containing two mirrors, place a file system on it, and mount it to simulate application access. The mirrors are completely synchronized by a read-writeback thread.

```
# vxstat -g adg -r
# vxassist -g adg make vol 1g layout=mirror nmirror=2
# vxstat -g adg -f ab vol
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol                  0         0    0.0      1024   2097152   15.5
# mkfs -F vxfs /dev/vx/rdsk/adg/vol
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
```

Step 2 We attach another plex intended to become our snapshot plex. Since the new plex still does not contain valid volume data, VxVM starts an atomic copy synchronization thread. We must await complete synchronization, until the plex may be used for snapshot purposes.

```
# vxstat -g adg -r
# vxassist -g adg mirror vol
# vxstat -g adg -f ab vol
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol               1024   2097152   12.9         0         0    0.0
```

Step 3 A frozen copy of volume data might be achieved by offlining one plex. But an offlined plex is twofold unavailable: its offline state prevents VxVM from reading and writing to the plex, and there is no device driver to this plex enabling application access. So, we dissociate the plex from the volume immediately stopping application I/O to it by the volume driver. Note the addition of the **-V** option to the plex dissociating command. It shows the basic **vxplex** command used for the volume usage type **fsgen** without actually dissociating the plex. We will come back to the meaning of the usage type.

```
# vxprint -rtg adg
[…]
v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
pl vol-03       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA
# vxplex -g adg -V dis vol-03
/usr/lib/vxvm/type/fsgen/vxplex -U fsgen -g adg -g adg -- dis vol-03
# vxplex -g adg dis vol-03
# vxprint -rtg adg
[…]
pl vol-03       -            DISABLED IOFAIL   2097152  CONCAT    -        RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA

v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
```

Step 4 A dissociated plex is still unavailable to an application due to its missing device driver. Only volumes provide a driver. Furthermore, a dissociated plex is forced to enter the kernel DISABLED state. In order to enable data availability, we add an empty volume to the dissociated plex and start both, plex and volume as well.

```
# vxmake -g adg vol SNAP-vol plex=vol-03 usetype=fsgen
# vxvol -g adg start SNAP-vol
# vxprint -rtg adg
[…]
v  SNAP-vol     -            ENABLED  ACTIVE   2097152  ROUND     -        fsgen
pl vol-03       SNAP-vol     ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA

v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
```

```
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1  ENA
```

Step 5 We dissociated the plex while the associated volume was mounted. In real life, you normally create a snapshot on a running application. Consequently, the data set presented by a snapshot volume is inconsistent from the application's point of view. In our example, the volume usetype **fsgen** forced the execution of a special **vxplex** command (see option **-V** above) triggering the file system layer to flush dirty kernel memory pages to the volume before dissociating it. Nevertheless, we expect our file system data set being inconsistent at least for one reason: the file system is marked as ACTIVE by the main super block simply due to its mounted state. So, we issue a file system check command, which will run really fast by replaying the file system log. However, our example working on Storage Foundation 5.0 MP1 does not need a file system check. Note that most software and patch versions of SF automatically issue a file system check after snapshot creation or plex dissociation.

```
# fsck -F vxfs -y /dev/vx/rdsk/adg/SNAP-vol
file system is clean - log replay is not required
```

Step 6 The snapshot volume may be mounted now and, for instance, used by backup tools. Afterwards, you could want to delete the snapshot volume. Our example demonstrates the steps necessary to reattach the snapshot to its original volume. First, we stop the snapshot volume and dissociate its plex once again, but this time from its snapshot volume. The empty volume should be deleted to prevent error messages from inadvertent volume access (as long as the volume exists as a standard volume, there is a driver on it).

```
# mkdir /mnt_snap
# mount -F vxfs /dev/vx/dsk/adg/SNAP-vol /mnt_snap
# df -k /mnt*
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/vx/dsk/adg/vol  1048576   17749  966408     2%    /mnt
/dev/vx/dsk/adg/SNAP-vol
                     1048576   17749  966408     2%    /mnt_snap
[…]
# umount /mnt_snap
# vxvol -g adg stop SNAP-vol
# vxplex -g adg dis vol-03
# vxedit -g adg rm SNAP-vol
# vxprint -rtg adg
[…]
pl vol-03       -            DISABLED -        2097152  CONCAT    -       RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2  ENA

v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -       fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -       RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0  ENA
```

```
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
```

Step 7 Reattaching the plex to its original volume means incrementing the number of data mirrors within the volume. Since our "orphaned" plex contains stale application data (to an amount unknown to VxVM), a full atomic copy synchronization thread is inevitable.

```
# vxstat -g adg -r
# vxplex -g adg att vol vol-03
# vxstat -g adg -f ab vol
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol               1024   2097152   12.7         0         0    0.0
# vxprint -rtg adg
[…]
v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -       fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
pl vol-03       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA
```

9.6 Legacy Snapshot Commands

The powerful **vxsnap** command was introduced in VxVM 4.0 to simplify administration of former snapshot mechanisms and especially to manage the new snapshot concepts: full sized instant snapshot and space optimized instant snapshot with shared cache volume. But **vxsnap** did not and still does not cover all VxVM 3.x snapshot techniques. Especially the kernel based fast mirror resynchronization was dropped from the **vxsnap** capabilities in favor to an exclusively DCO based snapshot architecture. Therefore, it is still worth to explain pre-**vxsnap** snapshot techniques and their command line interface.

9.6.1 Full Sized Snapshot without FMR

In the previous section, we already introduced the basic snapshot procedure of a so-called third mirror break-off. We recall that the snapshot was not instantly available, that it needed full resynchronization when reattaching it to its original volume (no fast mirror resynchronization), that it took a 100% volume size portion of storage, and that an immediate refresh by keeping the separate snapshot volume was impossible.

In VxVM 3.0, a new **vxassist** subtool was implemented to serve as an easy to handle interface to that snapshot procedure.

1. Adding a mirror-plex for snapshot purposes (a simple **vxassist mirror** command, see

step 2 above; we assume the twofold mirrored application volume already created), is replaced by **vxassist snapstart**. In order to verify the amount of synchronization I/Os, we reset the kernel I/O counters of VxVM and display atomic copy I/Os, as usual:

```
# vxstat –g adg -r
# vxassist -g adg snapstart vol
# vxstat -g adg -f a vol
                      ATOMIC COPIES
TYP NAME               OPS    BLOCKS AVG(ms)
vol vol               1024   2097152   12.7
# vxprint -rtg adg
[…]
v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
pl vol-03       vol          ENABLED  SNAPDONE 2097152  CONCAT    -        WO
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA
```

Note two small differences to the basic mirror tool: The new plex is marked for snapshot purposes by its application state SNAPDONE to tell the next step (creating a snapshot) which plex to dissociate. Furthermore, its mode is restricted to WO which stands for write-only: New volume data will keep the snapshot prepared plex up-to-date, but this plex, in most cases only a temporary member of the volume, will not modify the regular volume read policy.

2. Steps 3 to 4 (plex dissociation, volume frame, volume start), in most VxVM versions also step 5 (automatic file system check) of the previous chapter are replaced by:

```
# vxassist -g adg snapshot vol
# vxprint -rtg adg
[…]
v  SNAP-vol     -            ENABLED  ACTIVE   2097152  ROUND     -        fsgen
pl vol-03       SNAP-vol     ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA

v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
```

If you do not agree with the snapshot volume name automatically created by VxVM, you may specify it at snapshot creation:

```
# vxassist -g adg snapshot vol svol
```

3. Having terminated our duties with the snapshot volume, we might decide to reattach the snapshot plex to its original volume (see steps 6 and 7 above):

```
# vxstat –g adg -r
# vxassist -g adg snapback SNAP-vol
# vxstat -g adg -f a vol
                      ATOMIC COPIES
TYP NAME               OPS    BLOCKS AVG(ms)
vol vol               1024   2097152   12.8
# vxprint -rtg adg
[…]
v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
pl vol-03       vol          ENABLED  SNAPDONE 2097152  CONCAT    -        WO
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA
```

The command recreates the volume layout as it was at the end of step 1. Even the plex state is SNAPDONE again, and read access to it is prohibited. We did not need to specify the source volume. Obviously, somewhere, the snapshot relation between application volume and its snapshot volume was kept. But where? No snap objects are shown by **vxprint**, even by displaying all attributes (options **-a** or **-m**). The relation was stored in the kernel memory of VxVM. Thus, a system reboot or a disk group deport would have destroyed the snapshot relation.

4. Step 3 could be replaced by other procedures, which we will mention in few words. If you want to redirect the resynchronization, that is, from the snapshot plex to the original volume (don't forget to terminate application access), add the appropriate option:

```
# vxassist -g adg –o resyncfromreplica snapback SNAP-vol
```

After a system reboot or a disk group deport and re-import, the snapshot volume looks like and is indeed a volume completely independent from its former source volume. Advertently clearing the snapshot relation between both volumes does not require, of course, disk group deport and import, simply issue:

```
# vxassist –g adg snapclear vol [SNAP-vol]
```

Removing a snapshot volume does not differ from deleting a standard volume, in spite of the snapshot relation. Use one of the following commands:

```
# vxedit -g adg -rf rm SNAP-vol
# vxassist -g adg remove volume SNAP-vol
```

9.6.2 Full Sized Snapshot with Kernel Based FMR

The simple snapshot mechanism lacks a very important feature. Even though in many cases only a small percentage of data has changed between the original and the snapshot volume (either by writing to the original volume or to the snapshot volume), all volume data are synchronized when reattaching the snapshot plex. The advanced snapshot techniques explained in the main parts of this chapter use a DCL volume linked to the application volume by a DC object to track changed regions in a bitmap. In VxVM 3.1, another way to log modified regions of the volumes (original and snapshot) was introduced: a bitmap within the kernel memory. Well, we already know, that memory based region tracking is lost in case of a system reboot or disk group deport. But, not to forget an advantage, a kernel memory based bitmap does not degrade the performance of an application volume.

Either kernel or DCL volume based bitmap: We must tell the volume that we want to activate fast mirror resynchronization (FMR). The volume attribute **fastresync** must be set before dissociating the snapshot plex from its volume. For an already existing volume, enter:

```
# vxprint -g adg -F %fastresync vol
off
# vxvol -g adg set fastresync=on vol
# vxprint -g adg -F %fastresync vol
on
```

To set the **fastresync** attribute at volume creation time, issue:

```
# vxassist -g adg make vol 1g layout=mirror nmirror=2 fastresync=on
```

The following example, once again, demonstrates the effect of the kernel FMR bitmap. We modify application AND snapshot volume by 10 MB, but at non-overlapping regions. Thus, we expect synchronization of only 20 MB totally.

```
# vxassist -g adg make vol 1g layout=mirror nmirror=2 fastresync=on \
  init=active
# vxassist -g adg snapstart vol
# vxassist -g adg snapshot vol
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol bs=1024k count=10
# dd if=/dev/zero of=/dev/vx/rdsk/adg/SNAP-vol bs=1024k count=10 oseek=10
# vxstat -g adg -r
# vxassist -g adg snapback SNAP-vol
# vxstat -g adg -f a vol
                      ATOMIC COPIES
```

```
TYP NAME               OPS    BLOCKS AVG(ms)
vol vol                 20     40960   13.0
```

9.6.3 Full Sized Snapshot with DCL Volume Based FMR Version 0

The DCO structure is not an invention of VxVM 4.0, though this software version extended the DCO capabilities. Its basic task in VxVM 3.2 was to allow for fast mirror resynchronization in case of a snapback operation by persistently storing the required region bitmap in a DCL volume, not in the kernel memory, thus enabling offhost processing combined with the simultaneously introduced "Disk Group Split and Join" (DGSJ) feature. Adding DCO capabilities to a volume was a three-steps procedure with unique sequence: First, add the DCO structure, then enable FMR on the volume, and finally create the snapshot plex (or convert an existing plex to a snapshot plex). See the commands in detail:

```
# vxassist -g adg addlog vol logtype=dco
# vxvol -g adg set fastresync=on vol
# vxassist -g adg snapstart vol
# vxprint -rLtg adg
[…]
v  vol          -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol-01       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol-01       adg01    0        2097152  0         c1t1d0   ENA
pl vol-02       vol          ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol-02       adg02    0        2097152  0         c1t1d1   ENA
pl vol-03       vol          ENABLED  SNAPDONE 2097152  CONCAT    -        WO
sd adg03-01     vol-03       adg03    0        2097152  0         c1t1d2   ENA
dc vol_dco      vol          vol_dcl

v  vol_dcl      -            ENABLED  ACTIVE   144      SELECT    -        gen
pl vol_dcl-01   vol_dcl      ENABLED  ACTIVE   144      CONCAT    -        RW
sd adg01-02     vol_dcl-01   adg01    2097152  144      0         c1t1d0   ENA
pl vol_dcl-02   vol_dcl      ENABLED  ACTIVE   144      CONCAT    -        RW
sd adg02-02     vol_dcl-02   adg02    2097152  144      0         c1t1d1   ENA
pl vol_dcl-03   vol_dcl      DISABLED DCOSNP   144      CONCAT    -        RW
sd adg03-02     vol_dcl-03   adg03    2097152  144      0         c1t1d2   ENA
```

Replace the last command, if you want to mark an existing plex for snapshot purposes, by the following:

```
# vxplex -g adg -o dcoplex=vol_dcl-03 convert state=SNAPDONE vol-03
```

If you start from scratch, you may specify the first two snapshot related steps at volume creation time:

```
# vxassist -g adg make vol 1g layout=mirror,log nmirror=2 fastresync=on \
  logtype=dco
```

The **snapshot** and **snapback** commands are identical to the earlier snapshot techniques. Note the unusual small size of the DCL volume compared to the advanced **vxsnap** created DCL volume. This makes a difference to be explained.

9.7 DCO Version 0 and Version 20

The data change object linking the DCL volume to its application volume provides some interesting details. Issue the following command first on a legacy DC object created by vxassist addlog, then on a vxsnap built DC object:

```
# vxassist -g adg make vol00 1g layout=mirror,log nmirror=2 fastresync=on \
  logtype=dco init=active
# vxassist -g adg make vol20 1g layout=mirror,log nmirror=2 init=active
# vxsnap -g adg prepare vol20
# vxprint -g adg -m vol00_dco > /tmp/dco00
# vxprint -g adg -m vol20_dco > /tmp/dco20
# sdiff -w 80 /tmp/dco*
dco vol00_dco                          |  dco vol20_dco
[…]
        parent_vol=vol00               |        parent_vol=vol20
        log_vol=vol00_dcl              |        log_vol=vol20_dcl
        comment="DCO for vol00         |        comment="DCO for vol20
[…]
        p_flag_move=off                         p_flag_move=off
        badlog=off                              badlog=off
[…]
        sp_num=0                                sp_num=0
        regionsz=0                     |        regionsz=128
        version=0                      |        version=20
        drl=off                        |        drl=on
        sequentialdrl=off                       sequentialdrl=off
        drllogging=off                 |        drllogging=on
        snap=                                   snap=
```

Besides the object names and the record IDs skipped in the output above, we notice three major differences: the version number (0 and 20), the configurable region size and the ability to serve as dirty region log in version 20.

Let's start by examining the last feature. As we already know, a DRL is intended to track region changes in a mirrored volume for a certain amount of time in order to speed up resynchronization after a system crash. We will, by all means, just simulate a system crash. But nevertheless, be sure to carry out the following procedure in a test environment and to unmount all non-OS file systems except for our test volumes beforehand. Console access is a prerequisite.

```
# mkfs -F vxfs /dev/vx/rdsk/adg/vol00
# mkfs -F vxfs /dev/vx/rdsk/adg/vol20
# mkdir /mnt00 /mnt20
# mount -F vxfs /dev/vx/dsk/adg/vol00 /mnt00
# mount -F vxfs /dev/vx/dsk/adg/vol20 /mnt20
# vxprint -g adg -F '%name %devopen' vol00 vol20
vol00 on
vol20 on
# uadmin 5 0
panic[cpu513]/thread=300046b4b20: forced crash dump initiated at user request
[…]
dumping to /dev/dsk/c0t2d0s1, offset 215220224, content: kernel
[…]
ok boot -s
[…]
Requesting System Maintenance Mode
SINGLE USER MODE

Root password for system maintenance (control-d to bypass): password
# vxprint -rLtg adg
[…]
v  vol00        -            ENABLED  NEEDSYNC 2097152  SELECT    -        fsgen
pl vol00-01     vol00        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol00-01     adg01    0        2097152  0         c1t1d0   ENA
pl vol00-02     vol00        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-01     vol00-02     adg02    0        2097152  0         c1t1d1   ENA
dc vol00_dco    vol00        vol00_dcl

v  vol00_dcl    -            ENABLED  NEEDSYNC 144      SELECT    -        gen
pl vol00_dcl-01 vol00_dcl    ENABLED  ACTIVE   144      CONCAT    -        RW
sd adg01-02     vol00_dcl-01 adg01    2097152  144      0         c1t1d0   ENA
pl vol00_dcl-02 vol00_dcl    ENABLED  ACTIVE   144      CONCAT    -        RW
sd adg02-02     vol00_dcl-02 adg02    2097152  144      0         c1t1d1   ENA


v  vol20        -            ENABLED  NEEDSYNC 2097152  SELECT    -        fsgen
pl vol20-01     vol20        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-03     vol20-01     adg01    2097296  2097152  0         c1t1d0   ENA
pl vol20-02     vol20        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg02-03     vol20-02     adg02    2097296  2097152  0         c1t1d1   ENA
dc vol20_dco    vol20        vol20_dcl

v  vol20_dcl    -            ENABLED  NEEDSYNC 544      SELECT    -        gen
pl vol20_dcl-01 vol20_dcl    ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-04     vol20_dcl-01 adg01    4194448  544      0         c1t1d0   ENA
pl vol20_dcl-02 vol20_dcl    ENABLED  ACTIVE   544      CONCAT    -        RW
```

```
sd adg02-04     vol20_dcl-02 adg02    4194448  544      0         c1t1d1   ENA
# vxprint -g adg -F '%name %devopen' vol00 vol20
vol00 off
vol20 off
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol00                0         0    0.0         0         0    0.0
vol vol00_dcl            0         0    0.0         1       144   10.0
vol vol20                0         0    0.0         0         0    0.0
vol vol20_dcl            0         0    0.0         1       544   10.0
# vxstat -g adg -r
# exit
svc.startd: Returning to milestone all.
[…]
```

Please be patient until the boot process at the end of the legacy run-level 2 (**vxvm-recover**) has started the volume recovery of the OS volumes, before it turns to application volumes. During the early stage of the boot process, only the DCL volumes were synchronized.

```
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol00                0         0    0.0     16384   2097152    1.3
vol vol00_dcl            0         0    0.0         2       144    5.0
vol vol20                0         0    0.0         0         0    0.0
vol vol20_dcl            0         0    0.0         9       592    0.0
```

Indeed! Volume vol20 furnished with a DC object of version 20 did not synchronize the data volume because we had not written data to it immediately before the system crash (in case of I/O just a very small portion of the volume would have been synchronized). On the other side, the DC object of version 0 obviously does not provide dirty region logging, it has been completely resynchronized. Adding the legacy DRL plex to the application volume would cover this task.

We cannot answer the question probably arising why the DCL volumes were synchronized twice, the first time during the single-user mode (**vxvm-startup2**), the second time during the general volume resynchronization (**vxvm-recover**). Twofold synchronization is harmless to data consistency and, given the small size of the DCL volumes, means a system load you do not need to bother about.

Do you remember that a data plex attached by the **vxsnap addmir** command got the state pair ENABLED/SNAPDONE, while the attached DCL plex got DISABLED/DCOSNP as long as the snapshot is not performed? Well, the data plex must remain ENABLED, otherwise it would not be kept up-to-date. But the DCL plex attached for snapshot purposes may not be updated for dirty region log or temporary plex detach tasks, because we already have two active DCL plexes providing sufficient redundancy. Therefore, the DCL plex designed to be broken off together with the snapshot data plex got the DISABLED state to avoid unnecessary DCL plex I/O.

DC objects of version 0 or 20 just track changes to a volume in case of a snapshot plex break-off depending on the software version and the license you installed. An enterprise license implements another feature we all were waiting a long time for: optimized synchronization in case of temporary disk outage still keeping the volume enabled due to healthy data plexes. Assume a Dual data center scenario with volumes neatly mirrored over both sites. Furthermore, assume a temporary power failure at one site. The applications will continue to produce new data, but only on the remaining site. After powering back the failed site, the mirrors just differ to a certain amount of data (maybe 5%). The (fully licensed) DC object kept track of write changes to the volumes during the plexes' detach and will resynchronize just the affected regions. Regarding the technical behavior, a DC object version 0 differs only slightly from that of version 20.

```
# dd if=/dev/rdsk/c1t1d1s2 of=/var/tmp/c1t1d1s2 bs=128k iseek=1 count=8
# dd if=/dev/zero of=/dev/rdsk/c1t1d1s2 bs=128k oseek=1 count=8
# vxconfigd -k
# vxdisk -g adg list
DEVICE       TYPE            DISK         GROUP        STATUS
c1t1d0s2     auto:cdsdisk    adg01        adg          online
-            -          adg02        adg          failed was:c1t1d1s2
# vxprint -rLtg adg
[…]
v  vol00        -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol00-01     vol00        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-01     vol00-01     adg01    0        2097152  0         c1t1d0   ENA
pl vol00-02     vol00        DISABLED NODEVICE 2097152  CONCAT    -        RW
sd adg02-01     vol00-02     adg02    0        2097152  0         -        NDEV
dc vol00_dco    vol00        vol00_dcl

v  vol00_dcl    -            ENABLED  ACTIVE   144      SELECT    -        gen
pl vol00_dcl-01 vol00_dcl    ENABLED  ACTIVE   144      CONCAT    -        RW
sd adg01-02     vol00_dcl-01 adg01    2097152  144      0         c1t1d0   ENA
pl vol00_dcl-02 vol00_dcl    DISABLED NODEVICE 144      CONCAT    -        RW
sd adg02-02     vol00_dcl-02 adg02    2097152  144      0         -        NDEV


v  vol20        -            ENABLED  ACTIVE   2097152  SELECT    -        fsgen
pl vol20-01     vol20        ENABLED  ACTIVE   2097152  CONCAT    -        RW
sd adg01-03     vol20-01     adg01    2097296  2097152  0         c1t1d0   ENA
pl vol20-02     vol20        DISABLED NODEVICE 2097152  CONCAT    -        RW
sd adg02-03     vol20-02     adg02    2097296  2097152  0         -        NDEV
dc vol20_dco    vol20        vol20_dcl

v  vol20_dcl    -            ENABLED  ACTIVE   544      SELECT    -        gen
pl vol20_dcl-01 vol20_dcl    ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-04     vol20_dcl-01 adg01    4194448  544      0         c1t1d0   ENA
```

```
pl vol20_dcl-02 vol20_dcl    DISABLED NODEVICE 544      CONCAT    -        RW
sd adg02-04     vol20_dcl-02 adg02    4194448  544      0         -        NDEV
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol00 bs=1024k count=10
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol20 bs=1024k count=10
# dd if=/var/tmp/c1t1d1s2 of=/dev/rdsk/c1t1d1s2 bs=128k oseek=1
# vxdisk scandisks
# vxdg -g adg -k adddisk adg02=c1t1d1
# vxstat -g adg -r
# vxrecover -g adg
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol00               11     22528   23.6         0         0    0.0
vol vol00_dcl            1       144    0.0         0         0    0.0
vol vol20               10     20480   19.0         0         0    0.0
vol vol20_dcl            1       544   10.0         0         0    0.0
```

Tracking changes of volume data when a mirror is temporarily unavailable is not only useful in case of temporary disk outage. In order to keep a frozen volume data set, you do not need to go back to the somewhat oversized snapshot functionality. Just set one plex within the volume to OFFLINE state. Look at the following procedure to resynchronize the offlined plex to the current volume data set in case you want to continue with it. Only changed regions are synchronized.

```
# vxmend -g adg off vol00-02 vol20-02
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol00 bs=1024k count=10
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol20 bs=1024k count=10
# vxstat -g adg -r
# vxmend -g adg on vol00-02 vol20-02
# vxrecover -g adg
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol00               11     22528   20.9         0         0    0.0
vol vol00_dcl            0         0    0.0         0         0    0.0
vol vol20               10     20480   20.0         0         0    0.0
vol vol20_dcl            0         0    0.0         0         0    0.0
```

If you need to fall back to the frozen application data set:

```
# vxmend -g adg off vol00-02 vol20-02
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol00 bs=1024k count=10
# dd if=/dev/zero of=/dev/vx/rdsk/adg/vol20 bs=1024k count=10
# vxvol -g adg stop vol00 vol20
# vxmend -g adg on vol00-02 vol20-02
# vxmend -g adg fix stale vol00-01 vol20-01
# vxmend -g adg fix clean vol00-02 vol20-02
# vxstat -g adg -r
# vxrecover -g adg -s
# vxstat -g adg -f ab
                      ATOMIC COPIES               READ-WRITEBACK
TYP NAME               OPS    BLOCKS AVG(ms)      OPS    BLOCKS AVG(ms)
vol vol00               11     22528   20.9         0         0    0.0
vol vol00_dcl            0         0    0.0         0         0    0.0
vol vol20               10     20480   17.0         0         0    0.0
vol vol20_dcl            0         0    0.0         0         0    0.0
```

9.8 VxFS Storage Checkpoint Behavior

One of the most remarkable strengths of a VxFS storage checkpoint is its capability to avoid copy-on-first-writes in favor of just a slightly modified metadata set. We already mentioned that under specific circumstances VxFS switches to copy-on-first-writes, and we mentioned as well reasonable causes for that behavior. Let's test and discuss that topic!

We need four different types of write I/O operations onto a file system:

1. A file to be deleted (**Delete.1k**)
2. A file to be replaced by (new) content (**Replace.1k**)
3. A file to be enlarged (**Append.1k5**)
4. A file to be written by databases (**DBIO.10m**; the file remains at the same position keeping the same size, but some blocks within it are replaced)

```
# vxassist -g adg make vol 128m
# mkfs -F vxfs /dev/vx/rdsk/adg/vol
# mount -F vxfs /dev/vx/dsk/adg/vol /mnt
# cd /tmp
# mkfile 1k Delete.1k
# mkfile 1k Replace.1k
# mkfile 3b Append.1k5
# mkfile 10m DBIO.10m
# cp Delete.1k Replace.1k Append.1k5 DBIO.10m /mnt
# ls -l /mnt
total 20488
-rw-------   1 root     root     10485760 Sep 21 10:43 DBIO.10m
-rw-------   1 root     root        1536 Sep 21 10:43 Append.1k5
-rw-------   1 root     root        1024 Sep 21 10:43 Delete.1k
-rw-------   1 root     root        1024 Sep 21 10:43 Replace.1k
drwxr-xr-x   2 root     root          96 Sep 21 10:42 lost+found
# mount -F vxfs -o remount /dev/vx/dsk/adg/vol /mnt
# ncheck -F vxfs -o sector= /dev/vx/rdsk/adg/vol
[…]
UNNAMED     999      4    -       - 0/3202-0/3203 /Delete.1k
UNNAMED     999      5    -       - 0/3204-0/3205 /Replace.1k
UNNAMED     999      6    -      - 0/3208-0/3211 /Append.1k5
UNNAMED     999      7    -      - 0/16384-0/36863 /DBIO.10m
[…]
# fsckptadm create CKPT /mnt
# rm /mnt/Delete.1k
# cp /tmp/Replace.1k /mnt
# cat /tmp/Append.1k5 >> /mnt/Append.1k5
# perl -e '
  $Block="x" x 8192;
  open(FH,"+< /mnt/DBIO.10m") || die;
  sysseek(FH,81920,0);
  syswrite(FH,$Block,8192,0);
  close(FH);'
# mount -F vxfs -o remount /dev/vx/dsk/adg/vol /mnt
# ncheck -F vxfs -o sector= /dev/vx/rdsk/adg/vol
[…]
UNNAMED     999      5    -      - 0/5696-0/5697 /Replace.1k
UNNAMED     999      6    -      - 0/5698-0/5699 /Append.1k5
UNNAMED     999      6    -      - 0/3208-0/3211 /Append.1k5
UNNAMED     999      7    -      - 0/16384-0/36863 /DBIO.10m
CKPT       1000      4    -      - 0/3202-0/3203 /Delete.1k
CKPT       1000      5    -      - 0/3204-0/3205 /Replace.1k
CKPT       1000      6    -      - 0/3214-0/3215 /Append.1k5
CKPT       1000      7    -      - 0/5712-0/5727 /DBIO.10m
[…]
# mount -F vxfs -o ckpt=CKPT /dev/vx/dsk/adg/vol:CKPT /mnt_ckpt
# ls -l /mnt*
/mnt:
total 20488
-rw-------   1 root     root     10485760 Sep 21 10:50 DBIO.10m
-rw-------   1 root     root         3072 Sep 21 10:50 Append.1k5
-rw-------   1 root     root         1024 Sep 21 10:50 Replace.1k
drwxr-xr-x   2 root     root           96 Sep 21 10:42 lost+found

/mnt_ckpt:
total 22
-rw-------   1 root     root     10485760 Sep 21 10:43 DBIO.10m
-rw-------   1 root     root         1536 Sep 21 10:43 Append.1k5
-rw-------   1 root     root         1024 Sep 21 10:43 Delete.1k
-rw-------   1 root     root         1024 Sep 21 10:43 Replace.1k
drwxr-xr-x   2 root     root           96 Sep 21 10:42 lost+found
```

Examining the output of the **ncheck** and **ls** commands (especially the sector numbers and the time stamps), we conclude:

1. The deleted file content of **Delete.1k** remains at the same location within the file system (3202-3203), but is now addressed only by the checkpoint metadata. No copy-on-first-write!

2. The original data set of the overwritten, replaced file **Replace.1k** remains as well at the same location within the file system (3204-3205, 10:43), but visible only to the checkpoint after being overwritten. The new data blocks of the new file version (5696-5697, 10:50) visible to the active file system did not overwrite the previous version. No copy-on-first-write!
3. The blocks used to store the two versions of the file **Append.1k5** display a somewhat tricky, but quite intelligent behavior. Recall that, except for very large file systems, the default block size of VxFS is 1 kB. So, storing 1.5 kB of the original file **Append.1k5** allocated two file system blocks at 1 kB size each (sector numbers 3208-3211).

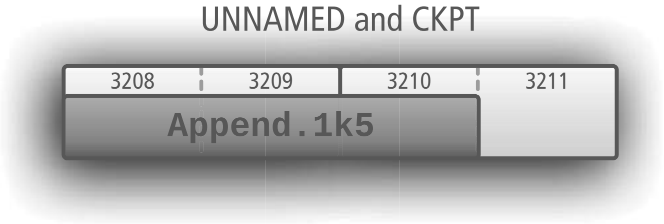


Figure 9-11: VxFS blocks of **Append.1k5** before appending data

Appending another 1.5 kB to this file enlarges the same file (inode number 6 remains unchanged) to a size of 3 kB. The two file system blocks of the original file (sectors 3208-3211, the last sector was previously unused) are still assigned to the active file system, so the first 512 bytes of the appended data are stored conveniently in the unused sector of the second 1 kB block. For the last 1 kB of the appended data a new file system block at a quite distant location (sectors 5698-5699) was allocated by the UNNAMED instance.

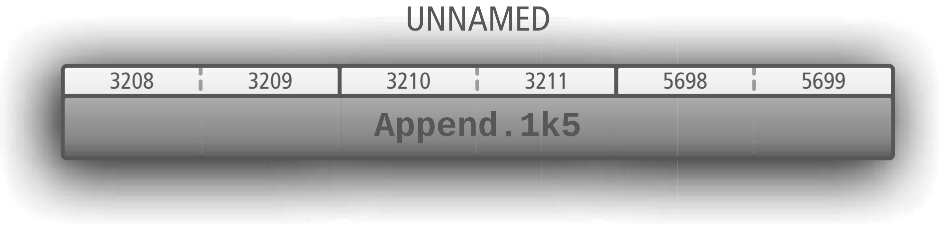


Figure 9-12: UNNAMED VxFS blocks of **Append.1k5** with appended data

The content of the second file system block of **Append.1k5** in its original state was copied to another location (sectors 3214-3215) and mapped by the checkpoint metadata, while the first block completely unmodified remains visible through the active and the checkpoint file system at the same time (not shown by the **ncheck** output).

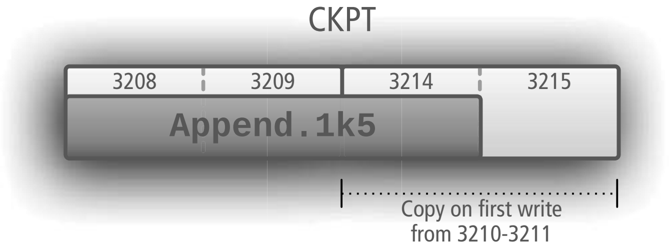


Figure 9-13: CKPT VxFS blocks of **Append.1k5** (data appended in UNNAMED)

The command **fsckptadm** provides an interface to track block changes and displays block allocations by the file system instances:

```
# fsckptadm blockinfo /mnt/Append.1k5 Ckpt /mnt
/mnt/Append.1k5:        <offset, len, flag>
                <0k, 1k, >
                <1k, 0k, CHANGED>
                <1k, 0k, EXTENDED>
                <2k, 1k, EXTENDED>
# fsckptadm blockinfo /mnt_ckpt/Append.1k5 Ckpt /mnt
/mnt_ckpt/Append.1k5:   <offset, len, flag>
                <0k, 1k, >
                <1k, 0k, CHANGED>
```

It is indeed quite difficult to generate a satisfactory output even by executing **fsckptadm blockinfo**. The second file system block (**offset** of **1k**) was actually extended by 512 bytes which is in case of an integer division indeed 0 kB (**len** of **0k**), while the first half of the block (512 B rounded down to 0 kB) effected a copy-on-first-write event (**<1k, 0k, CHANGED>**).

To sum up: Extending a file system block invokes a copy-on-first-write in favor of a preferably unfragmented active file (in spite of the fragmented allocation of the third block).

4. Based on our experience with the latter file, we assume a comparable block allocation policy in case of database-like I/O: The old block will be copied to another location, before the new data will be written to the original block position, thus keeping the active database file unfragmented. Our assumption is proved correct by a detailed analysis of the output of the **ncheck** command above and the following **fsckptadm** command:

```
# fsckptadm blockinfo /mnt/DBIO.10m Ckpt /mnt
/mnt/DBIO.10m:  <offset, len, flag>
```

```
                <0k, 80k, >
                <80k, 8k, CHANGED>
                <88k, 10152k, >
# fsckptadm blockinfo /mnt_ckpt/DBIO.10m Ckpt /mnt
/mnt_ckpt/DBIO.10m:     <offset, len, flag>
                <0k, 80k, >
                <80k, 8k, CHANGED>
                <88k, 10152k, >
```

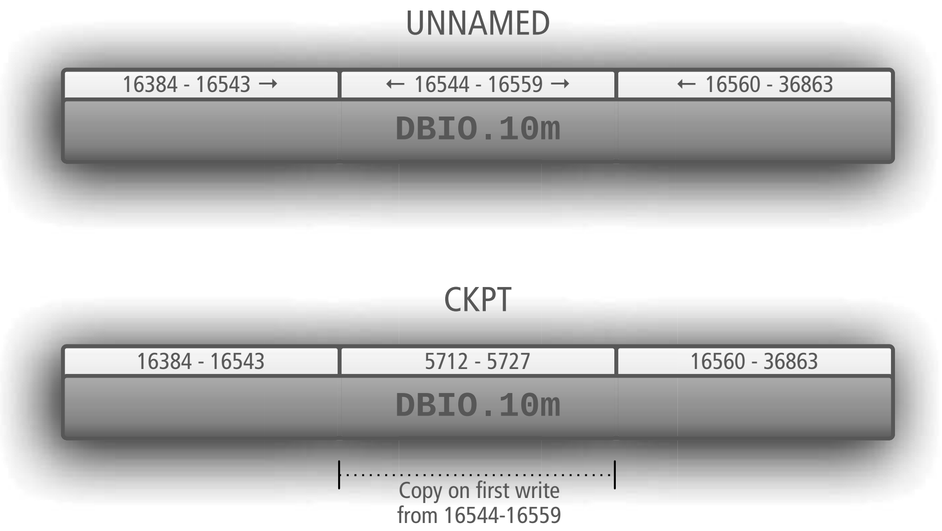


Figure 9-14: VxFS block allocation in case of database I/O