Btrfs Subvolume Quota Groups

Arne Jansen, Strato AG

October 2011

1 Subvolume Quota Concepts

The concept of quota has a long standing tradition in the unix world. Ever since computers allow multiple users to work simultaneously in one filesystem, there's the need to prevent one user from using up the full space. Every user should get his fair share of the available resources.

In case of files the solution is quite straightforward. Each file has an 'owner' recorded along with it, and it has a size. Traditional quota just restricts the total size of all files that are owned by a user. The concept is quite flexible: if a user hits his quota limit, the administrator can raise it on the fly.

On the other hand the traditional approach has only a poor solution to restrict directories. At installation time the harddisk can be partitioned so that every directory (e.g. /usr, /var, ...) that needs a limit gets its own partition. The obvious problem is, that those limits can't be changed without a reinstall. The btrfs subvolume feature builds a bridge. Subvolumes correspond in many ways to partitions as every subvolume looks like its own filesystem. With subvolume quota, it is now possible to restrict each subvolume like a partition, but keep the flexibility of quota. The space for each subvolume can be expanded or restricted on the fly.

As subvolumes are the basis for snapshots, interesting questions arise as to how to account used space in the presence of snapshots. If you have a file shared between a subvolume and a snapshot, whom to account the file to? The creator? Both? What if the file gets modified in the snapshot, should only these changes be accounted to it? But wait, both the snapshot and the subvolume belong to the same user home. I just want to limit the total space used by both! But somebody else might not want to charge the snapshots to the users.

Btrfs subvolume quota solves these problems by introducing groups of subvolumes and let the user put limits on them. It is even possible to have groups of groups. In the following we refer to them as "qgroups". Each qgroup primarily tracks two numbers, the amount of total referenced space and the amount of exclusively referenced space.

Referenced space is the amount of data that can be reached from any of the subvolumes contained in the qgroup, while *exclusive* is the amount of data where all references to this data can be reached from within this qgroup.

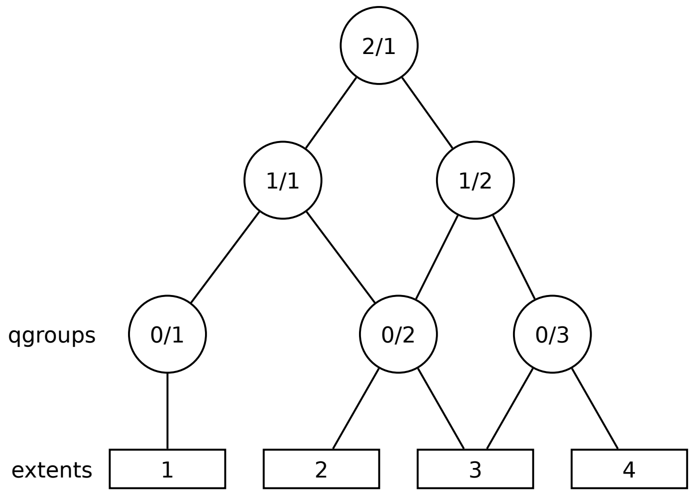

Figure 1: Sample qgroup hierarchy

2 Subvolume Quota Groups

The basic notion of the Subvolume Quota feature is the qouta group, short qgroup. Qgroups are notated as <level>/<id>, e.g. the qgroup 3/2 is a qgroup of level 3. For level 0 the leading “0/” can be omitted. Qgroups of level 0 get created automatically when a subvolume/snapshot gets created. The ID of the qgroup corresponds to the ID of the subvolume, so 0/5 is the qgroup for the root subvolume. For the “btrfs qgroup” command, the path to the subvolume can also be used instead of 0/<ID>. For all higher levels, the ID can be choosen freely.

Each qgroup can contain a set of lower level qgroups, thus creating a hierarchy of qgroups. Figure 1 shows an example qgroup tree. At the bottom some extents are depicted showing which qgroups reference which extents. It is important to understand the notion of *referenced* versus *exclusive*. In the example qgroup 0/2 references extents 2 and 3, while 1/2 references extents 2-4. 2/1 references all extents.

On the other hand, extent 1 is exclusive to 0/1, extent 2 is exclusive to 0/2, while extent 3 is neither exclusive to 0/2 nor to 0/3. But because both references can be reached from 1/2, extent 3 is exclusive to 1/2. All extents are exclusive to 2/1. So *exclusive* doesn’t mean there’s no other way to reach the extent, but it does mean that if you delete all subvolumes contained in a qgroup, the extent will get deleted. *Exclusive* of a qgroup conveys the useful information how much space will be freed in case all subvolumes of the qgroup get deleted.

All data extents are accounted this way. Metadata that belong to a specific subvolume (i.e. its fs tree) are also accounted. Checksums and extent allocation information are not accounted.

In turn the *referenced* count of a qgroup can be limited. All writes beyond that limit will lead to a “Quota Exceeded” error.

3 Inheritance

Things get a bit more complicated when new subvolumes or snapshots are created. The case of (empty) subvolumes is still quite easy. If a subvolume should be part of a qgroup, it has to be added to the qgroup at creation time. To add it at a later time, it would be necessary to at least rescan the full subvolume for a proper accounting.

Creation of a snapshot is the hard case. Obviously the snapshot will reference the exact amount of space as its source, and both source and destination now have an *exclusive* count of 0 (4k to be precise, as the roots of the trees are not shared). But what about qgroups of higher levels? If the qgroup contains both the source and the destination, nothing changes. If the qgroup contains only the source, it might lose some *exclusive*. But how much? The tempting answer is, “subtract all *exclusive* of the source from the qgroup”, but that’s wrong, or at least not enough. There could have been an extent that’s referenced from the source and another subvolume from that qgroup. This extent would have been exclusive to the qgroup, but not to the source subvolume. With the creation of the snapshot the qgroup would also lose this extent from its exclusive set.

So how can this problem be solved? In the instant the snapshot gets created we already have to know the correct *exclusive* count. We need to have a second qgroup that contains all the subvolumes as the first qgroup, except the subvolume we want to snapshot. The moment we create the snapshot, the *exclusive* count from the second qgroup needs to be copied to the first qgroup, as it represents the correct value. The second qgroup is called a tracking qgroup. It is only there in case a snapshot is needed.

4 Use Cases

4.1 single-user machine

4.1.1 Replacement for partitions

The simplest use case is to use qgroups as simple replacement for partitions. Btrfs takes the disk as a whole, and /, /usr, /var etc. are created as subvolumes. As each subvolume gets it own qgroup automatically, they can simply be restricted. No hierarchy is needed for that.

4.1.2 Track usage of snapshots

When a snapshot is taken, a qgroup for it will automatically be created with the correct values. *Referenced* will show how much is in it, possibly shared with other subvolumes. *Exclusive* will be the amount of space that gets freed when the subvolume is deleted.

4.2 Multi-user machine / Hosting

4.2.1 Restricting homes

When you have several users on a machine, with home dirs probably under /home, you might want to restrict /home as a whole, while restricting every user to an indiviual limit. This is easily accomplished by creating a qgroup for /home, e.g. 1/1, and assigning all user-subvols to it. Restricting this qgroup will limit /home, while every user subvol can get it's own (lower) limit.

4.2.2 Accounting snapshots to the user

Let's say the user is allowed to create snapshots via some mechanism. It would only be fair to account space used by the snapshots to the user. This doesn't mean the user doubles his usage as soon as he takes a snapshot. Of course files that are present in his home and the snapshot should only be accounted once. This can be accomplished by creating a qgroup for each user, say 1/<uid>. The user home and all snapshots are assigned to this qgroup. Limiting it will extend the limit to all snapshots, counting files only once. To limit /home as a whole, a higher level group 2/1 replacing 1/1 from the previous example is needed, with all user-qgroups assigned to it.

4.2.3 Don't account snapshots

On the other hand, when the snapshots get created automatically, the user has no chance to control them, so the space used by them should not be accounted to him. This is already the case when creating snapshots in example 4.2.1

4.2.4 Snapshots for backup purposes

This szenario is a mixture of the previous two. The user can create snapshots, but some snapshots for backup purposes are being created by the system. User's snapshots should be accounted to the user, system's not. The solution is similar to 4.2.2, but don't assign system snapshots to users qgroup.

5 Implementation

5.1 Update algorithm

The update algorithm is the core of the quota implementation. Whenever a reference is added or removed, the update algorithm is called.

The algorithm is called with the address of the extent for which to add/remove the reference, the root of the reference, the amount of space to add/remove, and of course the operation to perform.

A call could look like this

```
qgroup_record_ref(ref_root, start, num_bytes, operation);
```

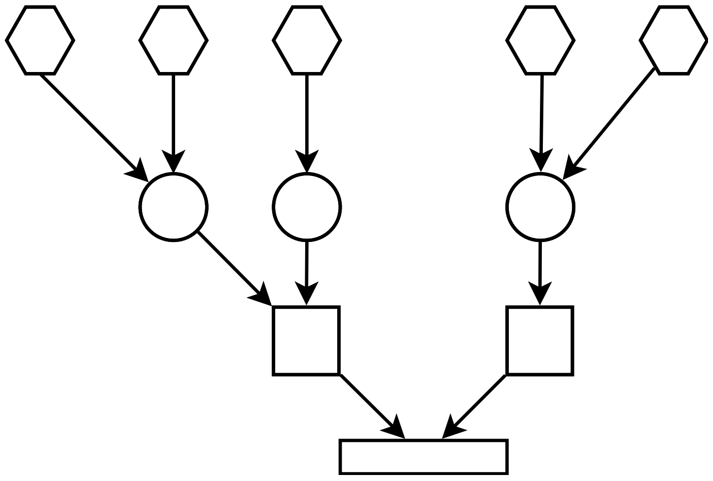

Figure 2: Extent with lazy references

In fact these parameters are all contained in the delayed ref structure, so just the delayed ref node is passed instead. This function gets called from the central point where backrefs are added to the filesystem, btrfs_add_delayed_*_ref.

The algorithm works in multiple steps:

1. Find all referencing roots
2. Calculate refcnt for all qgroups
3. Tag qgroups
4. Update *exclusive*

5.1.1 Find all referencing roots

The first step is to find all roots that are currently referencing the extent. Though btrfs is fully back-referenced, this step is not as easy as it may seem, because of the lazy refcounting scheme. The back references that are recorded for the extent may not tell the full truth. In figure 2 a tree is depicted where the actual extent only has 2 back references recorded, whereas there are 5 roots referencing it.

The solution is to walk up the tree and follow all back references until all roots are found. This looks like a classic problem for a recursive tree walk, but recursion here is not possible for two reasons:

1. The code runs in kernel space with very limited stack space. With a recursion the stack may overflow.
2. To follow a back reference, the referenced extent has to be searched. This is due to the nature of the indirect back references used. These back references point to a key in the tree, not to the address of an extent.

The code solves this by keeping two lists, one for all roots found and one for all backrefs to follow. Initially, the list of roots is empty, while the list of backrefs is filled with only one item, the reference to the extent for which all backrefs are to be found.

The following pseudo-code describes how all roots are found:

```
foreach ref (0 ... #refs in ulist)
    find extent for ref
    add all refs for extent to ulist
    if (extent is root)
        add root to ulist of roots
```

The lists here are called 'ulists', because they only accept new items if they are not already in the list, i.e. if they are unique.

The step to add all backrefs for an extent involves finding all recorded inline backrefs, all in-tree backrefs and all delayed refs for the extent up to the moment the algorithms starts to run. Because this code might run some time, new delayed refs for any extent in the tree might be added in the meantime. To avoid a race condition here, each delayed ref gets a sequence number. Only delayed refs with seq $<$ own seq are considered. Also, no delayed ref with a higher seq than own seq must be run while the roots are searched for.

The code will never include the reference to add/delete.

5.1.2 Calculate refcnt for all qgroups

After the list of referencing roots is known, the next 3 steps all operate on the qgroup hierarchy. A sample hierarchy is depicted in figure 1.

The first operation on the tree is to calculate the number of references that can be reached from every given qgroup. This is done by walking the tree upwards from every root found in the previous step and incrementing a count on each qgroup visited, where each root can only increment the count by one for every qgroup it can reach, even if it can reach it by several paths. The calculated count is called the refcnt.

As in the previous step, the tree is walked iteratively with the help of ulists to avoid recursion. Figure 4 depicts the state after this step is done for extent 3, where the ref from 0/2 should get deleted. The Figure omits the fs-trees and their roots, as qgroups of level 0 directly correspond to a root.

As the refcnt is part of the qgroup struct, the algorithm would require that all refcnts in all qgroups be set to zero before it can run. To avoid this, a global sequence number is used to determine the refcnt. Only one thread at a time can currently do refcounting on the tree (that is easily changed should it impose a limit). This thread grabs the next sequence number and walks up the tree. If the refcnt of the visited qgroup is smaller than the seq, it is not yet set and known to be 0. Otherwise, it is incremented. After the algorithm has run, the global sequence number is incremented by the max refcnt found.

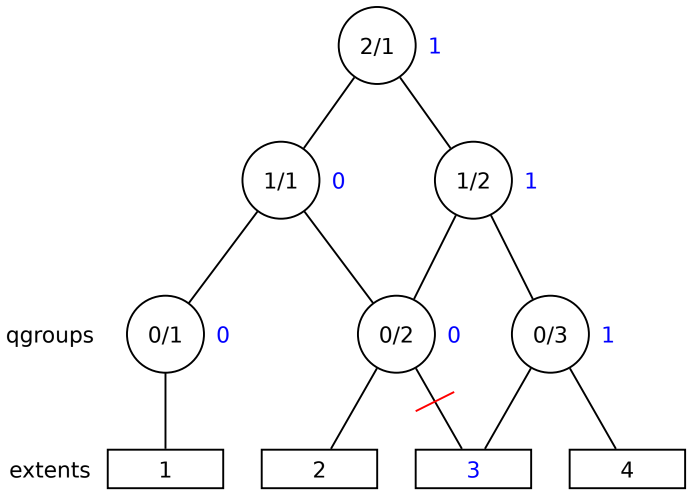

Figure 3: qgroup tree after refcnt augmentation for extent 3

5.1.3 Tag qgroups

The next step is to walk up the tree again, but this time starting with ref_root, the root to add/remove. Remember that the previous step doesn't include the ref_root. Every qgroup that is being visited on the way up will be tagged in preparation for the next step. Additionally, under certain conditions a first adjustment is made to the values of the visited qgroups.

- If the refcnt is zero and the operation is to add a reference, this means this qgroup is not yet referencing this extent, but after the operation it will, so the *referenced* value of the qgroup is increased by num_bytes.
- If the refcnt is zero and the operation is to remove a reference, this means this qgroup is currently referencing the extent, but through the operation it will lose its last reference, so the *referenced* value is decreased by num_bytes.
- If the refcnt is zero and the number of roots found in the first step is also zero, this means:
- in case of addition: the added reference will be the only reference, so *exclusive* of the qgroup is increased by num_bytes
- in case of removal: the reference is the last to remove, which means it is currently exclusive to ref_root, so *exclusive* of the qgroup is decreased by num_bytes

Figure 4 depicts the situation given the reference for 0/2 to extent 3 is to be deleted. *Referenced* of 0/2 and 1/1 will get decreased.

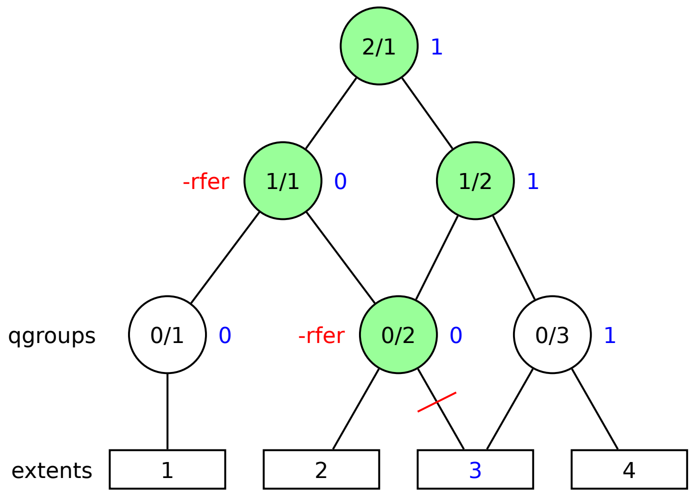

Figure 4: qgroup tree after tagging

5.1.4 Update *exclusive*

The last step adjusts the *exclusive* counts of all untagged qgroups. The *exclusive* counts of the tagged qgroups already got adjusted in the previous step. All roots from step 1 are walked again, tagged qgroups are skipped. If the refcnt equals the number of roots found in step one, *exclusive* gets increased if the ref is to be removed and decreased otherwise. Figure 5 shows the outcome of this step. Extent 3 is now *exclusive* to 0/3. All other *exclusives* are untouched. Extent 3 was exclusive to 1/2 and 2/1 and still is, while it wasn't exclusive to 0/2 and 1/1 and still isn't.

5.2 Tracking Groups

As seen in the introductory chapter, when taking a snapshot the values of several qgroups might need to be adjusted. This is easiest to see when looking at some examples. Figure 6 shows a simple example where tracking groups are needed.

The exercise is to track *referenced* and *exclusive* for all snapshots of a subvolume. The gray qgroups 0/2-0/4 are all snapshot of 0/1. Before 0/4 is created, 1/2 contains 0/2 and 0/3. The moment 0/4 gets created, it is added to 1/2. The *exclusive* count of 1/2 will not change, as all extents that become reachable from 1/2 are also reachable from 1/1. More problematic is the *referenced* count, as not all extents from 0/4 might be new to 1/2. The solution is to add another qgroup, 1/3, that tracks 0/1 and all subvolumes of it (Figure 7).

The moment the snapshot gets created, 1/3 holds the correct *referenced* count for all snapshots. To get 1/2 back to the correct values, *referenced* from 1/3 has to be copied to 1/2, while *exclusive* of 1/2 stays untouched.

In the next step, we want to take a snapshot of 0/2. The resulting snapshot

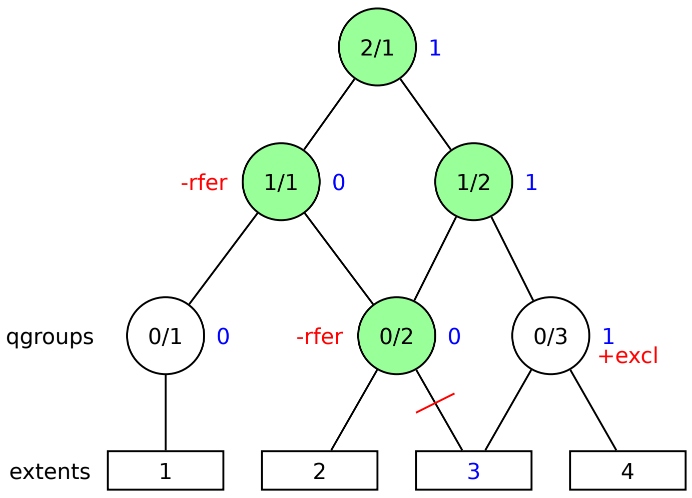

Figure 5: update of *exclusive* on qgroup tree

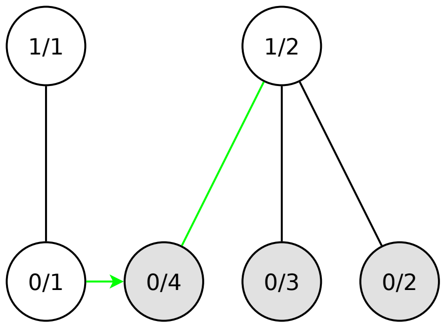

Figure 6: Tracking snapshots

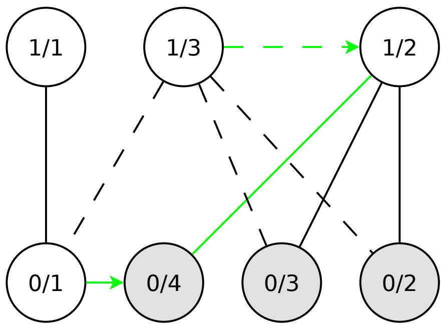

Figure 7: A tracking qgroup is needed for 1/2

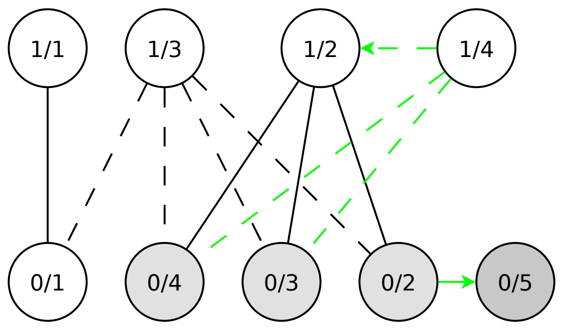

Figure 8: A snapshot of 0/2

should not be part of 1/2. This poses another problem: while *referenced* does not change, *exclusive* needs to be corrected. For this, we need another tracking group, 1/4 (Figure 8).

When 0/5 is created, *exclusive* from 1/4 needs to be copied to 1/2. Snapshotting 0/2 also invalidates the *exclusive* of 1/3. Also, another snapshot of 0/1 would invalidate 1/4. So one more tracking groups is needed, containing 0/1, 0/4 and 0/3.

It is planned that the btrfs userland utility will keep track of the needed tracking groups and takes care that all the necessary copies happen. For this, a format needs to be found how a user can describe what snapshots he intends to take. Keeping tracking groups for all possible combinations would lead to an exponential number of tracking groups.

5.3 On-disk quota tree layout

Qgroups add a new tree, the quota tree. 4 new keys are used in this tree. The overall status is recorded in a status item, each qgroup has 2 items, one to record the user configured limits and one to record the current *referenced*/*exclusive* counts. Each parent/child-relationship between qgroups gets 2 qgroup_relation items, one per direction. The on-disk structure is still preliminary.

```
/*
 * Records the overall state of the qgroups.
 * There's only one instance of this key present,
 * (0, BTRFS_QGROUP_STATUS_KEY, 0)
 */
#define BTRFS_QGROUP_STATUS_KEY 240
/*
 * Records the currently used space of the qgroup.
 * One key per qgroup, (0, BTRFS_QGROUP_INFO_KEY, qgroupid).
 */
#define BTRFS_QGROUP_INFO_KEY 242
```

```
/*
 * Contains the user configured limits for the qgroup.
 * One key per qgroup, (0, BTRFS_QGROUP_LIMIT_KEY, qgroupid).
 */
#define BTRFS_QGROUP_LIMIT_KEY 244
/*
 * Records the child-parent relationship of qgroups. For
 * each relation, 2 keys are present:
 * (childid, BTRFS_QGROUP_RELATION_KEY, parentid)
 * (parentid, BTRFS_QGROUP_RELATION_KEY, childid)
 */
#define BTRFS_QGROUP_RELATION_KEY 246
```

The keys are chosen in a way that first comes the STATUS_KEY, followed by all INFO_KEYs, followed by all LIMIT_KEYs. After that, for each qgroup present all relations follow. Only the INFO_KEYs and the STATUS_KEY get updated regularly. The idea is that those keys stay close to each other, to minimize writes. The RELATION_KEY is chosen in a way that by a simple enumeration all children and parents for a given qgroup can be found. The qgroupid is composed of a 16 bit 'level' field, followed by a 48 bit 'id' field. A qgroupid is represented as level/id, e.g. 2/100. In the case of a subvolume, the level is 0, and the 'id' is just the internal tree objectid (5 or >= 256). On the command line, the user will be able to use the subvolume-path as the identifier.

```
/*
 * is subvolume quota turned on?
 */
#define BTRFS_QGROUP_STATUS_FLAG_ON (1ULL << 0)
/*
 * SCANNING is set during the initialization phase
 */
#define BTRFS_QGROUP_STATUS_FLAG_SCANNING (1ULL << 1)
/*
 * Some qgroup entries are known to be out of date,
 * either because the configuration has changed in a way that
 * makes a rescan necessary, or because the fs has been mounted
 * with a non-qgroup-aware version.
 * Turning qouta off and on again makes it inconsistent, too.
 */
#define BTRFS_QGROUP_STATUS_FLAG_INCONSISTENT (1ULL << 2)
#define BTRFS_QGROUP_STATUS_VERSION 1
struct btrfs_qgroup_status_item {
    __le64 version;
    /*
     * the generation is updated during every commit. As older
     * versions of btrfs are not aware of qgroups, it will be
```

```
         * possible to detect inconsistencies by checking the
         * generation on mount time
         */
        __le64 generation;
        /* flag definitions see above */
        __le64 flags;
        /*
         * only used during scanning to record the progress
         * of the scan. It contains a logical address
         */
        __le64 scan;
    } __attribute__ ((__packed__));
```

Instead of hosting the scan cursor in the structure, one could also make a separate key instead that is only present during scanning.

```
    struct btrfs_qgroup_info_item {
        /*
         * only updated when any of the other values change
         */
        __le64 generation;
        __le64 rfer;
        __le64 rfer_cmpr;
        __le64 excl;
        __le64 excl_cmpr;
    } __attribute__ ((__packed__));
```

For all uncompressed data the same value will be recorded for compressed and uncompressed. The *_cmpr values represent the amount of disk space used, the other values the amount of space from a user perspective. The uncompressed values are hard to get, so a first version might not support them yet and just record the on-disk values instead.

```
    /* flags definition for qgroup limits */
    #define BTRFS_QGROUP_LIMIT_MAX_RFER (1ULL << 0)
    #define BTRFS_QGROUP_LIMIT_MAX_EXCL (1ULL << 1)
    #define BTRFS_QGROUP_LIMIT_RSV_RFER (1ULL << 2)
    #define BTRFS_QGROUP_LIMIT_RSV_EXCL (1ULL << 3)
    #define BTRFS_QGROUP_LIMIT_RFER_CMPR (1ULL << 4)
    #define BTRFS_QGROUP_LIMIT_EXCL_CMPR (1ULL << 5)
    struct btrfs_qgroup_limit_item {
        __le64 flags;
        __le64 max_referenced;
        __le64 max_exclusive;
        __le64 rsv_referenced;
        __le64 rsv_exclusive;
    } __attribute__ ((__packed__));
```

The flags record which of the limits are to be enforced. The last two flags indicate whether the compressed or the uncompressed value is to limit. This structure also contains reservations, though they might be hard to implement, as btrfs has no clear understanding of how much free space is left. A straightforward implementation might be very inaccurate and the first version will probably not implement it. Those values are nevertheless included here as a means for future expansion.

5.4 Estimation

In btrfs, each file operation is encapsulated into a transaction. All necessary space for the transaction has to be reserved before any modification is done to the structures, as there's no way to back out in the middle. That's what block reserves are used for.

The same holds for quota: it's not possible to deny an operation in the middle of it. The only point where an EDQUOT (Quota exceeded) error can be generated is before the operation start. The easiest way would be to only deny it if one of the affected qgroups is already over quota, but that would allow large operations to exceed the quota by far. This implementation tries to estimate the needed space for the operation and reserves it at operation start. If the reservation fails, the operation is denied.

The reservation is recorded in each qgroup. Also it is saved in the trans_handle, so it can be freed on end_transaction. The estimation is not a worst-case estimation like the block reservation. It shouldn't deny requests too early. On the other hand, it might be possible that a qgroup goes slightly over quota.