Elastic Quotas: Stretch Your Disk Space

Erez Zadok, Jeffrey Osborn, Ariye Shater,
Charles Wright, and Kiran-Kumar Muniswamy-Reddy
Stony Brook University

Jason Nieh
Columbia University

Abstract

Elastic quotas are a novel method of managing storage resources, which allows users to exceed their persistent quota while disk resources are available. When storage becomes scarce, the system automatically reclaims some storage based on preconfigured policies. Elastic quota policies are the core of the elastic quota system: they allow administrators and users to select the appropriate methods for reclaiming space. We have implemented a prototype of the elastic quota system in Linux, including a highly-configurable policy control system.

1 Introduction

Despite seemingly endless increases in the amount of storage and ever decreasing costs of hardware, managing storage is still expensive. Additionally, users continue to fill increasingly larger disks, worsened by the proliferation of large multimedia files and high-speed broadband networks.

Storage management has been a problem in a past, continues to be a problem today, and is only getting worse—all despite growing disk sizes.Our Elastic Quota system is designed to help the management problem via efficient allocation of storage while allowing users maximal freedom, all with minimal administrator intervention.

Elastic quotas enter users into an agreement with the system: users can exceed their quota while space is available, under the condition that the system will be able to automatically reclaim the storage when the need arises. Users or applications may designate some files as *elastic*. When space runs short, the *elastic quota system* (Equota) may reclaim space from those files marked as elastic; non-elastic files maintain existing semantics and are accounted for in users' persistent quotas. Storage reclamation can be accomplished through either compression, migration, or removal of the elastic file.

2 Design

Our two primary design goals were to allow for versatile and efficient elastic quota policy management. To achieve efficiency we designed the system to run as a kernel file system with DB3 [1] databases accessible to the user-level tools. We used a stackable file system to ensure we do not have to modify existing file systems such as Ext3 [2].

We mark a file as elastic using a single inode bit; such spare bits are available in most modern disk-based file systems such as FFS, UFS, and Ext2/3. To change elasticity, a standard `ioctl` toggles the bit. To avoid recursive scanning of all files in the file system, we use DB3 databases to associate a user ID, file name, and inode number. Using the DB3 databases helps to improve performance by avoiding recursive scanning. In the case of database corruption or deletion, the databases can be rebuilt from information from the lower-level file system.

System Operation In traditional operating systems, quota accounting is often integrated with the native disk-based file system. Since Linux supports a number of file systems with quotas, quota accounting is an independent VFS component called *dquot*. Usually, system calls invoke VFS operations which in turn call file-system–specific operations. However, unlike other VFS code, Dquot code does *not* call the file system. Instead, the native file system calls the Dquot operations directly via the Dquot operations vector, initialized when quotas are turned on for that file system. This reverse calling sequence is necessary since only the native disk-based file system knows when a user operation results in a change in consumption of inodes or disk blocks.

Our stackable file system EQFS intercepts file system operations, performs related elastic quota operations, and then passes the operation to the lower file system (Ext2, Ext3, etc.). EQFS also intercepts the system call used to turn on quota management and inserts its own set of quota management operations vector in the component which we call *edquot*. However, the calling convention for quota operations is reversed: from the lower file system, through our elastic quota management (Edquot), and to the VFS's own disk quota management (Dquot). This novel interception is a form of *reverse stacking* we devised to avoid changing Dquot (and hence the VFS) or native file systems.

Each user on our system has two UIDs: one that accounts for persistent quotas and another (the *shadow UID*) that accounts for elastic quotas. The shadow UID is simply the ones-complement of the former. It does not modify existing ownership or permissions semantics, it is only used for quota accounting. When the Edquot operations are called, Edquot determines if the operation was for an elastic or a persistent file, and informs dquot to account for the changed resource (inode or disk block) for either the UID or shadow UID. This allows us to use the existing quota infrastructure to account for elastic usage, and allows programs to get this information easily through the quota system.

Equota's database management and file system cleaning activities are both controlled by the daemon *Rubberd*. When certain file system calls are made, EQFS informs Rubberd of events which create or change the association of an inode to file name or owner. Rubberd records this information in the DB3 databases. EQFS informs Rubberd about creation, deletion, renames, hard links, and ownership changes of elastic files. EQFS communicates this information to Rubberd's database management thread over a Linux kernel-to-user socket called *netlink*.

Rubberd's database management thread listens for netlink messages from EQFS. When it receives a message, Rubberd decodes it and applies the proper operation on the per-user database. For example, when a file is made elastic, EQFS sends a "create elastic file" netlink message to Rubberd along with the UID, inode number, and the name of the file. Rubberd then inserts a new entry in that user's database, using the inode number as key and the file's name as the entry's value.

Rubberd is configured to wakeup periodically and record each user's historical elastic space utilization over a period of time. Rubberd gets the list of all users, their elastic and persistent disk usage, and their elastic and persistent quotas. With these numbers, Rubberd computes the average utilization of each user over a period of time, and stores this value in the database. The computed average is used by rubberd to allow for reclamation proportional across all users, allowing our system to fairly reclaim storage.

3 Elastic Quota Policies

The core of the elastic quota system is its handling of space reclamation policies. EQFS is the file system that provides elasticity support and works in conjunction with Rubberd, the user-space daemon that implements cleaning policies.

To the people involved, file system reclamation policies must consider three factors: fairness, convenience, and gaming. Fairness is considered when an administrator determines through a policy what the order of file reclamation should be. Since the idea of fairness varies from user to user, we provide flexibility in the policies and leave it to the administrator's discretion. Convenience in this case is the ease of use of the system. If a system isn't easy to use, it is often not used, and goes to waste. Finally, a policy must be resistant to gaming, or "cheating". Such a policy would include a number of criteria for reclamation that are not easily changed by the user, preventing them from cheating the system.

These three factors are important especially in light of efficiency, because some policies could be executed more efficiently than others. However, our overall design goals in this work were to provide as much flexibility to both administrators and users to decide on the suitable set of policies that meet their site's needs.

When the file system usage exceeds a high level watermark, rubberd computes the amount of space to be reclaimed– the *goal*, and reclaims space based on the cleaning policies chosen by the administrator. The administrator can specify multiple reclamation policies in a configuration file and rubberd then evaluates each policy in order until the goal is reached. A policy can be broken up into four parts: the type, the method, the sort order and an optional filter.

type defines the kind of policy to use and it can have one three values: `global` where all the elastic files in eqfs are considered irrespective of the owner of the file; `user` where in files owned by each each user is evaluated independently; and `user_policy` which is a user mode where the user has his own policy file for space reclamation. A `user_policy` is a file stored in `/var/spool/rubberd/`*USER* that consists of a policy file including the user's reclamation order preferences.

method specifes the action to be taken to reclaim space and it can be `rm` that specifies files must be deleted; `gzip` which specifies that files must be compressed; `mv` and `tar` can be used to migrate files to a slower media.

sort specifies the order in which files must be reclaimed. It can be `size` which specifies that larger files must be deleted first, `mtime` the oldest modified files must be deleted first; and similarly, `atime` and `ctime`.

An optional filter can be specified for global and user policies that defines the file names or extensions to which the policies must be applied.

When rubberd determines that space must be reclaimed, it computes the goal blocks to be recovered from each user and deletes files accordingly. The amount of space that should be recovered from each user can be proportional to the current or historical elastic space utilization of the user, as chosen by the administrator. The current utilization can be measured as: the amount of elastic space being used by a user, the amount of elastic quota that exceeds the unused persistant space of a user, the total persistant and elastic space being used by the user. The historical utilization can be measured either as linear average of usage over a period of time or the exponentially-decaying average.

4 Summary and Status

The main contribution of this paper is in the exploration and evaluation of various elastic quota policies, demonstrating the utility of treating storage as an elastic resource. Our Linux prototype includes many features that allow both site administrators and users to tailor their elastic quota policies to their needs. For example, we provide several different ways to decide when a file becomes elastic: from the directory's mode, from the file's name, from the user's login session, and even by the application itself. Our policy engine is flexible, allowing a variety of methods for elastic space reclamation. Our evaluation shows that the performance overheads are small and acceptable for day-to-day use. We observed an overhead of 1.5% when compiling `gcc`. Finally, our work provides an extensible framework for new or custom policies to be added.

We plan to expand upon the definition of persistent and elastic files to include a file lifetime. A file lifetime would include a minimum and maximum lifetime. A persistent file has an infinite minimum lifetime and an elastic file has a minimum lifetime of zero. The minimum lifetime would be useful for data that may not be relevant for longer than some predefined time period. A maximum lifetime would provide for the automatic deletion of files; this could be valuable in scenarios such as automatic enforcement of a company's records retention policy or automatic deletion of personal information after a certain point. This type of lifetime would also serve as a better priority for space reclamation than atime or mtime.

Finally, we plan to deploy Equota on an instrumented large multi-user system. Combined with user surveys, this will provide us with more insight into how end-users utilize Equota in an real-world setting.

References

[1] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. http://www.sleepycat.com.

[2] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.