S3QL Documentation

Release 1.13.1

Nikolaus Rath

February 24, 2013

CONTENTS

CHAPTER

ONE

ABOUT S3QL

S3QL is a file system that stores all its data online using storage services like Google Storage, Amazon S3 or OpenStack. S3QL effectively provides a hard disk of dynamic, infinite capacity that can be accessed from any computer with internet access running Linux, FreeBSD or OS-X.

S3QL is a standard conforming, full featured UNIX file system that is conceptually indistinguishable from any local file system. Furthermore, S3QL has additional features like compression, encryption, data de-duplication, immutable trees and snapshotting which make it especially suitable for online backup and archival.

S3QL is designed to favor simplicity and elegance over performance and feature-creep. Care has been taken to make the source code as readable and serviceable as possible. Solid error detection and error handling have been included from the very first line, and S3QL comes with extensive automated test cases for all its components.

1.1 Features

- **Transparency.** Conceptually, S3QL is indistinguishable from a local file system. For example, it supports hardlinks, symlinks, standard unix permissions, extended attributes and file sizes up to 2 TB.
- **Dynamic Size.** The size of an S3QL file system grows and shrinks dynamically as required.
- **Compression.** Before storage, all data may compressed with the LZMA, bzip2 or deflate (gzip) algorithm.
- **Encryption.** After compression (but before upload), all data can AES encrypted with a 256 bit key. An additional SHA256 HMAC checksum is used to protect the data against manipulation.
- **Data De-duplication.** If several files have identical contents, the redundant data will be stored only once. This works across all files stored in the file system, and also if only some parts of the files are identical while other parts differ.
- **Immutable Trees.** Directory trees can be made immutable, so that their contents can no longer be changed in any way whatsoever. This can be used to ensure that backups can not be modified after they have been made.
- **Copy-on-Write/Snapshotting.** S3QL can replicate entire directory trees without using any additional storage space. Only if one of the copies is modified, the part of the data that has been modified will take up additional storage space. This can be used to create intelligent snapshots that preserve the state of a directory at different points in time using a minimum amount of space.
- **High Performance independent of network latency.** All operations that do not write or read file contents (like creating directories or moving, renaming, and changing permissions of files and directories) are very fast because they are carried out without any network transactions.

  S3QL achieves this by saving the entire file and directory structure in a database. This database is locally cached and the remote copy updated asynchronously.

- **Support for low bandwidth connections.** S3QL splits file contents into smaller blocks and caches blocks locally. This minimizes both the number of network transactions required for reading and writing data, and the amount of data that has to be transferred when only parts of a file are read or written.

1.2 Development Status

After two years of beta-testing by about 93 users did not reveal any data-critical bugs, S3QL was declared **stable** with the release of version 1.0 on May 13th, 2011. Note that this does not mean that S3QL is bug-free. S3QL still has several known, and probably many more unknown bugs. However, there is a high probability that these bugs will, although being inconvenient, not endanger any stored data.

Please report any problems on the mailing list or the issue tracker.

CHAPTER

TWO

INSTALLATION

S3QL depends on several other programs and libraries that have to be installed first. The best method to satisfy these dependencies depends on your distribution. In some cases S3QL and all its dependencies can be installed with as little as three commands, while in other cases more work may be required.

The S3QL Wiki contains installation instructions for quite a few different Linux distributions. You should only use the generic instructions in this manual if your distribution is not included in the distribution-specific installation instructions on the wiki.

2.1 Dependencies

The following is a list of the programs and libraries required for running S3QL. Generally, you should first check if your distribution already provides a suitable packages and only install from source if that is not the case.

- Kernel: Linux 2.6.9 or newer or FreeBSD with FUSE4BSD. Starting with kernel 2.6.26 you will get significantly better write performance, so under Linux you should actually use *2.6.26 or newer whenever possible*.
- Python 2.7.0 or newer (but not Python 3.x). Make sure to also install the development headers.
- The PyCrypto++ Python Module. To check if this module is installed, try to execute `python -c 'import pycryptopp'`.
- SQLite version 3.7.0 or newer. SQLite has to be installed as a *shared library* with development headers.
- The APSW Python Module. To check which (if any) version of APWS is installed, run the command

  ```
  python -c 'import apsw; print apsw.apswversion()'
  ```

  The printed version number should be at least 3.7.0. Note that APSW must be linked *dynamically* against SQLite, so you can *not* use the Ubuntu PPA at https://launchpad.net/~ubuntu-rogerbinns/+archive/apsw (these packages are statically linked).
- The PyLibLZMA Python module. To check if this module is installed, execute `python -c 'import lzma; print lzma.__version__'`. This should print a version number. You need at least version 0.5.3.
- The Python LLFUSE module. To check if this module is installed, execute `python -c 'import llfuse; print llfuse.__version__'`. This should print a version number. You need at least version 0.37.

2.2 Installing S3QL

To install S3QL itself, proceed as follows:

1. Download S3QL from http://code.google.com/p/s3ql/downloads/list
2. Unpack it into a folder of your choice
3. Run `python setup.py build` to build S3QL.
4. Run `python setup.py test` to run a self-test. If this fails, ask for help on the mailing list or report a bug in the issue tracker.

Now you have three options:

- You can run the S3QL commands from the `bin/` directory.
- You can install S3QL system-wide for all users. To do that, you have to run `sudo python setup.py install`.
- You can install S3QL into `~/.local` by executing `python setup.py install --user`. In this case you should make sure that `~/.local/bin` is in your `$PATH` environment variable.

2.3 Development Version

If you have checked out the unstable development version from the Mercurial repository, a bit more effort is required. You need to also have Cython (0.16 or newer) and Sphinx (1.1 or newer) installed, and the necessary commands are:

```
python setup.py build_cython
python setup.py build_ext --inplace
python setup.py build_sphinx
python setup.py test
python setup.py install
```

CHAPTER

THREE

STORAGE BACKENDS

S3QL supports different *backends* to store data at different service providers and using different protocols. A *storage url* specifies a backend together with some backend-specific information and uniquely identifies an S3QL file system. The form of the storage url depends on the backend and is described for every backend below.

All storage backends respect the `http_proxy` and `https_proxy` environment variables.

3.1 Google Storage

Google Storage is an online storage service offered by Google. To use the Google Storage backend, you need to have (or sign up for) a Google account, and then activate Google Storage for your account. The account is free, you will pay only for the amount of storage and traffic that you actually use. Once you have created the account, make sure to activate legacy access.

To create a Google Storage bucket, you can use e.g. the Google Storage Manager. The storage URL for accessing the bucket in S3QL is then

```
gs://<bucketname>/<prefix>
```

Here *bucketname* is the name of the bucket, and *prefix* can be an arbitrary prefix that will be prepended to all object names used by S3QL. This allows you to store several S3QL file systems in the same Google Storage bucket.

Note that the backend login and password for accessing your Google Storage bucket are not your Google account name and password, but the *Google Storage developer access key* and *Google Storage developer secret* that you can manage with the Google Storage key management tool.

3.2 Amazon S3

Amazon S3 is the online storage service offered by Amazon Web Services (AWS). To use the S3 backend, you first need to sign up for an AWS account. The account is free, you will pay only for the amount of storage and traffic that you actually use. After that, you need to create a bucket that will hold the S3QL file system, e.g. using the AWS Management Console. For best performance, it is recommend to create the bucket in the geographically closest storage region, but not the US Standard region (see below).

The storage URL for accessing S3 buckets in S3QL has the form

```
s3://<bucketname>/<prefix>
```

Here *bucketname* is the name of the bucket, and *prefix* can be an arbitrary prefix that will be prepended to all object names used by S3QL. This allows you to store several S3QL file systems in the same S3 bucket.

Note that the backend login and password for accessing S3 are not the user id and password that you use to log into the Amazon Webpage, but the *AWS access key id* and *AWS secret access key* shown under My Account/Access Identifiers.

3.2.1 Reduced Redundancy Storage (RRS)

S3QL does not allow the use of reduced redundancy storage. The reason for that is a combination of three factors:

- RRS has a relatively low reliability, on average you lose one out of every ten-thousand objects a year. So you can expect to occasionally lose some data.
- When `fsck.s3ql` asks S3 for a list of the stored objects, this list includes even those objects that have been lost. Therefore `fsck.s3ql` *can not detect lost objects* and lost data will only become apparent when you try to actually read from a file whose data has been lost. This is a (very unfortunate) peculiarity of Amazon S3.
- Due to the data de-duplication feature of S3QL, unnoticed lost objects may cause subsequent data loss later in time (see *Data Durability* for details).

3.3 OpenStack/Swift

OpenStack is an open-source cloud server application suite. Swift is the cloud storage module of OpenStack. Swift/OpenStack storage is offered by many different companies.

The storage URL for the OpenStack backend has the form

```
swift://<hostname>[:<port>]/<container>[/<prefix>]
```

Note that the storage container must already exist. Most OpenStack providers offer a web frontend that you can use to create storage containers. *prefix* can be an arbitrary prefix that will be prepended to all object names used by S3QL. This allows you to store several S3QL file systems in the same container.

The OpenStack backend always uses HTTPS connections. Note, however, that at this point S3QL does not verify the server certificate (cf. issue 267).

3.4 RackSpace CloudFiles

RackSpace CloudFiles uses OpenStack internally, so you can use the OpenStack/Swift backend (see above). The hostname for CloudFiles containers is `auth.api.rackspacecloud.com`. Use your normal RackSpace user name for the backend login, and your RackSpace API key as the backend passphrase. You can create a storage container for S3QL using the Control Panel (go to *Cloud Files* under *Hosting*).

Warning: As of January 2012, RackSpace does not give any information about data consistency or data durability on their web page. However, RackSpace support agents (especially in the live chat) often claim very high guarantees. Any such statement is wrong. As of 01/2012, RackSpace CloudFiles does *not* give *any* durability or consistency guarantees (see *Important Rules to Avoid Losing Data* for why this is important). Why this fact is only acknowledged RackSpace's technical engineers, and/or not communicated to their sales agents is not known.

You should note that opinions about RackSpace differ widely among S3QL users and developers. On one hand, people praise RackSpace for their backing of the (open source) OpenStack project. On the other hand, their heavily advertised "fanatical support" is in practice often not only less than helpful, but their support agents also seem to be downright incompetent. However, there are reports that the support quality increases dramatically once you are a customer and use the "Live Chat" link when you are logged into the control panel.

3.5 S3 compatible

The S3 compatible backend allows S3QL to access any storage service that uses the same protocol as Amazon S3. The storage URL has the form

```
s3c://<hostname>:<port>/<bucketname>/<prefix>
```

Here *bucketname* is the name of an (existing) bucket, and *prefix* can be an arbitrary prefix that will be prepended to all object names used by S3QL. This allows you to store several S3QL file systems in the same bucket.

3.6 Local

S3QL is also able to store its data on the local file system. This can be used to backup data on external media, or to access external services that S3QL can not talk to directly (e.g., it is possible to store data over SSH by first mounting the remote system using **'sshfs'_** and then using the local backend to store the data in the sshfs mountpoint).

The storage URL for local storage is

```
local://<path>
```

Note that you have to write three consecutive slashes to specify an absolute path, e.g. `local:///var/archive`. Also, relative paths will automatically be converted to absolute paths before the authentication file (see *Storing Authentication Information*) is read, i.e. if you are in the `/home/john` directory and try to mount `local://s3ql`, the corresponding section in the authentication file must match the storage url `local:///home/john/s3ql`.

CHAPTER

FOUR

IMPORTANT RULES TO AVOID LOSING DATA

Most S3QL backends store data in distributed storage systems. These systems differ from a traditional, local hard disk in several important ways. In order to avoid losing data, this section should be read very carefully.

4.1 Rules in a Nutshell

To avoid losing your data, obey the following rules:

1. Know what durability you can expect from your chosen storage provider. The durability describes how likely it is that a stored object becomes damaged over time. Such data corruption can never be prevented completely, techniques like geographic replication and RAID storage just reduce the likelihood of it to happen (i.e., increase the durability).
2. When choosing a backend and storage provider, keep in mind that when using S3QL, the effective durability of the file system data will be reduced because of S3QL's data de-duplication feature.
3. Determine your storage service's consistency window. The consistency window that is important for S3QL is the smaller of the times for which:
   - a newly created object may not yet be included in the list of stored objects
   - an attempt to read a newly created object may fail with the storage service reporting that the object does not exist

   If *one* of the above times is zero, we say that as far as S3QL is concerned the storage service has *immediate* consistency.

   If your storage provider claims that *neither* of the above can ever happen, while at the same time promising high durability, you should choose a respectable provider instead.
4. When mounting the same file system on different computers (or on the same computer but with different `--cachedir` directories), the time that passes between the first and second of invocation of **mount.s3ql** must be at least as long as your storage service's consistency window. If your storage service offers immediate consistency, you do not need to wait at all.
5. Before running **fsck.s3ql** or **s3qladm**, the file system must have been left untouched for the length of the consistency window. If your storage service offers immediate consistency, you do not need to wait at all.

The rest of this section explains the above rules and the reasons for them in more detail. It also contains a list of the consistency windows for a number of larger storage providers.

4.2 Consistency Window List

The following is a list of the consistency windows (as far as S3QL is concerned) for a number of storage providers. This list doesn't come with any guarantees and may be outdated. If your storage provider is not included, or if you need more reliable information, check with your storage provider.

Storage Provider	Consistency
Amazon S3 in the US standard region	Eventual
Amazon S3 in other regions	Immediate
Google Storage	Immediate
RackSpace CloudFiles	Eventual

4.3 Data Consistency

In contrast to the typical hard disk, most storage providers do not guarantee *immediate consistency* of written data. This means that:

- after an object has been stored, requests to read this object may still fail or return the prior contents for a little while.
- after an object has been deleted, attempts to read it may still return the (old) data for some time, and it may still remain in the list of stored objects for some time.
- after a new object has been created, it may still not be included when retrieving the list of stored objects for some time.

Of course, none of this is acceptable for a file system, and S3QL generally handles any of the above situations internally so that it always provides a fully consistent file system to the user. However, there are some situations where an S3QL user nevertheless needs to be aware of the peculiarities of his chosen storage service.

Suppose that you mount the file system, store some new data, delete some old data and unmount it. If you then mount the file system again right away on another computer, there is no guarantee that S3QL will see any of the changes that the first S3QL process has made. At least in theory it is therefore possible that on the second mount, S3QL does not see any of the changes that you have done and presents you an "old version" of the file system without them. Even worse, if you notice the problem and unmount the file system, S3QL will upload the old status (which S3QL necessarily has to consider as current) and thereby permanently override the newer version (even though this change may not become immediately visible either). S3QL uses several techniques to reduce the likelihood of this to happen (see *Implementation Details* for more information on this), but without support from the storage service, the possibility cannot be eliminated completely.

The same problem of course also applies when checking the file system. If the storage service provides S3QL with only partially updated data, S3QL has no way to find out if this a real consistency problem that needs to be fixed or if it is only a temporary problem that will resolve itself automatically (because there are still changes that have not become visible yet).

This is where the so called *consistency window* comes in. The consistency window is the maximum time (after writing or deleting the object) for which any of the above "outdated responses" may be received. If the consistency window is zero, i.e. all changes are immediately effective, the storage service is said to have *immediate consistency*. If the window is infinite, i.e. there is no upper bound on the time it may take for changes to become effect, the storage service is said to be *eventually consistent*. Note that often there are different consistency windows for the different operations. For example, Google Storage offers immediate consistency when reading data, but only eventual consistency for the list of stored objects.

To prevent the problem of S3QL working with an outdated copy of the file system data, it is therefore sufficient to simply wait for the consistency window to pass before mounting the file system again (or running a file system check). The length of the consistency window changes from storage service to storage service, and if your service is not

included in the list below, you should check the web page or ask the technical support of your storage provider. The window that is important for S3QL is the smaller of the times for which

- a newly created object may not yet be included in the list of stored objects
- an attempt to read a newly created object may fail with the storage service reporting that the object does not exist

Unfortunately, many storage providers are hesitant to guarantee anything but eventual consistency, i.e. the length of the consistency window is potentially infinite. In that case you simply have to pick a length that you consider "safe enough". For example, even though Amazon is only guaranteeing eventual consistency, the ordinary consistency window for data stored in S3 is just a few seconds, and only in exceptional circumstances (i.e., core network outages) it may rise up to hours (source).

4.4 Data Durability

The durability of a storage service a measure of the average probability of a storage object to become corrupted over time. The lower the chance of data loss, the higher the durability. Storage services like Amazon S3 claim to achieve a durability of up to 99.999999999% over a year, i.e. if you store 100000000 objects for 100 years, you can expect that at the end of that time one object will be corrupted or lost.

S3QL is designed to reduce redundancy and store data in the smallest possible form. Therefore, S3QL is generally not able to compensate for any such losses, and when choosing a storage service you should carefully review if the offered durability matches your requirements. When doing this, there are two factors that should be kept in mind.

Firstly, even though S3QL is not able to compensate for storage service failures, it is able to detect them: when trying to access data that has been lost or corrupted by the storage service, an IO error will be returned and the mount point will become inaccessible to ensure that the problem is noticed.

Secondly, the consequences of a data loss by the storage service can be significantly more severe than you may expect because of S3QL's data de-duplication feature: a data loss in the storage service at time *x* may cause data that is written *after* time *x* to be lost as well. Consider the following scenario:

1. You store an important file in the S3QL file system.
2. The storage service loses the data blocks of this file. As long as you do not access the file or run **fsck.s3ql**, S3QL is not aware that the data has been lost by the storage service.
3. You save an additional copy of the important file in a different location on the same S3QL file system.
4. S3QL detects that the contents of the new file are identical to the data blocks that have been stored earlier. Since at this point S3QL is not aware that these blocks have been lost by the storage service, it does not save another copy of the file contents in the storage service but relies on the (presumably) existing blocks instead.
5. Therefore, even though you saved another copy, you still do not have a backup of the important file (since both copies refer to the same data blocks that have been lost by the storage service).

For some storage services, **fsck.s3ql** can mitigate this effect. When **fsck.s3ql** runs, it asks the storage service for a list of all stored objects. If objects are missing, it can then mark the damaged files and prevent the problem from spreading forwards in time. Figuratively speaking, this establishes a "checkpoint": data loss that occurred before running **fsck.s3ql** can not affect any file system operations that are performed after the check. Unfortunately, many storage services only "discover" that objects are missing or broken when the object actually needs to be retrieved. In this case, **fsck.s3ql** will not learn anything by just querying the list of objects.

In the future, **fsck.s3ql** will have an additional "full-check" mode, in which it attempts to retrieve every single object. However, this is expected to be rather time consuming and expensive. Therefore, it is generally a better choice to choose a storage service where the expected data durability is so high that the possibility of a lost object (and thus the need to run any full checks) can be neglected over long periods of time.

To some degree, **fsck.s3ql** can mitigate this effect. When used with the `--full-check` option, **fsck.s3ql** asks the storage service to look up every stored object. This way, S3QL learns about any missing and, depending on the storage service, corrupted objects. It can then mark the damaged files and prevent the problem from spreading forwards in time. Figuratively speaking, this establishes a "checkpoint": data loss that occurred before running **fsck.s3ql** with `--full-check` can not affect any file system operations that are performed after the check.

Unfortunately, a full check is rather time consuming and expensive because of the need to check every single stored object. It is generally a better choice to choose a storage service where the expected data durability is so high that the possibility of a lost object (and thus the need to run any full checks) can be neglected over long periods of time.

CHAPTER

FIVE

FILE SYSTEM CREATION

A S3QL file system is created with the `mkfs.s3ql` command. It has the following syntax:

```
mkfs.s3ql [options] <storage url>
```

This command accepts the following options:

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

-L <name> Filesystem label

--max-obj-size <size> Maximum size of storage objects in KiB. Files bigger than this will be spread over multiple objects in the storage backend. Default: 10240 KiB.

--plain Create unencrypted file system.

--force Overwrite any existing data.

Unless you have specified the `--plain` option, `mkfs.s3ql` will ask you to enter an encryption password. This password will *not* be read from an authentication file specified with the `--authfile` option to prevent accidental creation of an encrypted file system.

CHAPTER

SIX

MANAGING FILE SYSTEMS

The `s3qladm` command performs various operations on *unmounted* S3QL file systems. The file system *must not be mounted* when using `s3qladm` or things will go wrong badly.

The syntax is

```
s3qladm [options] <action> <storage-url>
```

where `action` may be either of **passphrase**, **upgrade**, **clear** or **download-metadata**.

The **s3qladm** accepts the following general options, no matter what specific action is being invoked:

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--log <target> Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `none`

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--version just print program version and exit

Hint: run `s3qladm <action> --help` to get help on the additional arguments that the different actions take.

6.1 Changing the Passphrase

To change the passphrase of a file system, use the `passphrase` subcommand:

```
s3qladm passphrase  <storage url>
```

6.2 Upgrading the file system

If you have installed a new version of S3QL, it may sometimes be necessary to upgrade the file system metadata as well. Note that in this case the file system can no longer be accessed with older versions of S3QL after the upgrade.

During the upgrade you have to make sure that the command is not interrupted, and that no one else tries to mount, check or upgrade the file system at the same time.

To upgrade a file system from the previous to the current revision, execute

```
s3qladm upgrade <storage url>
```

6.3 Deleting a file system

A file system can be deleted with:

```
s3qladm clear <storage url>
```

This physically deletes all the data and file system structures.

6.4 Restoring Metadata Backups

If the most-recent copy of the file system metadata has been damaged irreparably, it is possible to restore one of the automatically created backup copies.

The command

```
s3qladm download-metadata <storage url>
```

will give you a list of the available metadata backups and allow you to download them. This will create two new files in the current directory, ending in `.db` and `.params`. To actually use the downloaded backup, you need to move these files into the `~/.s3ql/` directory and run `fsck.s3ql`.

Warning: You should probably not use this functionality without having asked for help on the mailing list first (see *Further Resources / Getting Help*).

CHAPTER

SEVEN

MOUNTING

A S3QL file system is mounted with the `mount.s3ql` command. It has the following syntax:

```
mount.s3ql [options] <storage url> <mountpoint>
```

Note: S3QL is not a network file system like NFS or CIFS. It can only be mounted on one computer at a time.

This command accepts the following options:

--log <target> Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `~/.s3ql/mount.log`

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

--cachesize <size> Cache size in KiB (default: 102400 (100 MiB)). Should be at least 10 times the maximum object size of the filesystem, otherwise an object may be retrieved and written several times during a single write() or read() operation.

--max-cache-entries <num> Maximum number of entries in cache (default: 768). Each cache entry requires one file descriptor, so if you increase this number you have to make sure that your process file descriptor limit (as set with `ulimit -n`) is high enough (at least the number of cache entries + 100).

--allow-other Normally, only the user who called `mount.s3ql` can access the mount point. This user then also has full access to it, independent of

individual file permissions. If the `--allow-other` option is specified, other users can access the mount point as well and individual file permissions are taken into account for all users.

--allow-root Like `--allow-other`, but restrict access to the mounting user and the root user.

--fg Do not daemonize, stay in foreground

--single Run in single threaded mode. If you don't understand this, then you don't need it.

--upstart Stay in foreground and raise SIGSTOP once mountpoint is up.

--profile Create profiling information. If you don't understand this, then you don't need it.

--compress <name> Compression algorithm to use when storing new data. Allowed values: `lzma`, `bzip2`, `zlib`, none. (default: `lzma`)

--metadata-upload-interval <seconds> Interval in seconds between complete metadata uploads. Set to 0 to disable. Default: 24h.

--threads <no> Number of parallel upload threads to use (default: auto).

--nfs Enable some optimizations for exporting the file system over NFS. (default: False)

7.1 Compression Algorithms

S3QL supports three compression algorithms, LZMA, Bzip2 and zlib (with LZMA being the default). The compression algorithm can be specified freely whenever the file system is mounted, since it affects only the compression of new data blocks.

Roughly speaking, LZMA is slower but achieves better compression ratios than Bzip2, while Bzip2 in turn is slower but achieves better compression ratios than zlib.

For maximum file system performance, the best algorithm therefore depends on your network connection speed: the compression algorithm should be fast enough to saturate your network connection.

To find the optimal algorithm and number of parallel compression threads for your system, S3QL ships with a program called `benchmark.py` in the `contrib` directory. You should run this program on a file that has a size that is roughly equal to the block size of your file system and has similar contents. It will then determine the compression speeds for the different algorithms and the upload speeds for the specified backend and recommend the best algorithm that is fast enough to saturate your network connection.

Obviously you should make sure that there is little other system load when you run `benchmark.py` (i.e., don't compile software or encode videos at the same time).

7.2 Notes about Caching

S3QL maintains a local cache of the file system data to speed up access. The cache is block based, so it is possible that only parts of a file are in the cache.

7.2.1 Maximum Number of Cache Entries

The maximum size of the cache can be configured with the `--cachesize` option. In addition to that, the maximum number of objects in the cache is limited by the `--max-cache-entries` option, so it is possible that the cache does not grow up to the maximum cache size because the maximum number of cache elements has been reached. The reason for this limit is that each cache entry requires one open file descriptor, and Linux distributions usually limit the total number of file descriptors per process to about a thousand.

If you specify a value for `--max-cache-entries`, you should therefore make sure to also configure your system to increase the maximum number of open file handles. This can be done temporarily with the `umask -n` command. The method to permanently change this limit system-wide depends on your distribution.

7.2.2 Cache Flushing and Expiration

S3QL flushes changed blocks in the cache to the backend whenever a block has not been accessed for at least 10 seconds. Note that when a block is flushed, it still remains in the cache.

Cache expiration (i.e., removal of blocks from the cache) is only done when the maximum cache size is reached. S3QL always expires the least recently used blocks first.

7.3 Automatic Mounting

If you want to mount and umount an S3QL file system automatically at system startup and shutdown, you should do so with one dedicated S3QL init script for each S3QL file system.

If your system is using upstart, an appropriate job can be defined as follows (and should be placed in `/etc/init/`):

```
description        "S3QL Backup File System"
author                "Nikolaus Rath <Nikolaus@rath.org>"

# This assumes that eth0 provides your internet connection
start on (filesystem and net-device-up IFACE=eth0)

# We can't use "stop on runlevel [016]" because from that point on we
# have only 10 seconds until the system shuts down completely.
stop on starting rc RUNLEVEL=[016]

# Time to wait before sending SIGKILL to the daemon and
# pre-stop script
kill timeout 300

env STORAGE_URL="s3://my-backup-bla"
env MOUNTPOINT="/mnt/backup"

env USER="myusername"
env AUTHFILE="/path/to/authinfo2"

expect stop

script
    # Redirect stdout and stderr into the system log
    DIR=$(mktemp -d)
    mkfifo "$DIR/LOG_FIFO"
    logger -t s3ql -p local0.info < "$DIR/LOG_FIFO" &
    exec > "$DIR/LOG_FIFO"
```

```
    exec 2>&1
    rm -rf "$DIR"

    # Check and mount file system
    su -s /bin/sh -c 'exec "$0" "$@"' "$USER" -- \
        fsck.s3ql --batch --authfile "$AUTHFILE" "$STORAGE_URL"
    exec su -s /bin/sh -c 'exec "$0" "$@"' "$USER" -- \
        mount.s3ql --upstart --authfile "$AUTHFILE" "$STORAGE_URL" "$MOUNTPOINT"
end script

pre-stop script
    su -s /bin/sh -c 'exec "$0" "$@"' "$USER" -- umount.s3ql "$MOUNTPOINT"
end script
```

Note: In principle, it is also possible to automatically mount an S3QL file system with an appropriate entry in `/etc/fstab`. However, this is not recommended for several reasons:

- file systems mounted in `/etc/fstab` will be unmounted with the `umount` command, so your system will not wait until all data has been uploaded but shutdown (or restart) immediately (this is a FUSE limitation, see issue 159).
- There is no way to tell the system that mounting S3QL requires a Python interpreter to be available, so it may attempt to run `mount.s3ql` before it has mounted the volume containing the Python interpreter.
- There is no standard way to tell the system that internet connection has to be up before the S3QL file system can be mounted.

CHAPTER

EIGHT

ADVANCED S3QL FEATURES

8.1 Snapshotting and Copy-on-Write

The command `s3qlcp` can be used to duplicate a directory tree without physically copying the file contents. This is made possible by the data de-duplication feature of S3QL.

The syntax of `s3qlcp` is:

```
s3qlcp [options] <src> <target>
```

This will replicate the contents of the directory `<src>` in the directory `<target>`. `<src>` has to be an existing directory and `<target>` must not exist. Moreover, both directories have to be within the same S3QL file system.

The replication will not take any additional space. Only if one of directories is modified later on, the modified data will take additional storage space.

`s3qlcp` can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

Note that:

- After the replication, both source and target directory will still be completely ordinary directories. You can regard `<src>` as a snapshot of `<target>` or vice versa. However, the most common usage of `s3qlcp` is to regularly duplicate the same source directory, say `documents`, to different target directories. For a e.g. monthly replication, the target directories would typically be named something like `documents_January` for the replication in January, `documents_February` for the replication in February etc. In this case it is clear that the target directories should be regarded as snapshots of the source directory.
- Exactly the same effect could be achieved by an ordinary copy program like `cp -a`. However, this procedure would be orders of magnitude slower, because `cp` would have to read every file completely (so that S3QL had to fetch all the data over the network from the backend) before writing them into the destination folder.

8.1.1 Snapshotting vs Hardlinking

Snapshot support in S3QL is inspired by the hardlinking feature that is offered by programs like rsync or storeBackup. These programs can create a hardlink instead of copying a file if an identical file already exists in the backup. However, using hardlinks has two large disadvantages:

- backups and restores always have to be made with a special program that takes care of the hardlinking. The backup must not be touched by any other programs (they may make changes that inadvertently affect other hardlinked files)

- special care needs to be taken to handle files which are already hardlinked (the restore program needs to know that the hardlink was not just introduced by the backup program to safe space)

S3QL snapshots do not have these problems, and they can be used with any backup program.

8.2 Getting Statistics

You can get more information about a mounted S3QL file system with the `s3qlstat` command. It has the following syntax:

```
s3qlstat [options] <mountpoint>
```

Probably the most interesting numbers are the total size of your data, the total size after duplication, and the final size after de-duplication and compression.

`s3qlstat` can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

For a full list of available options, run `s3qlstat --help`.

8.3 Immutable Trees

The command **s3qllock** can be used to make a directory tree immutable. Immutable trees can no longer be changed in any way whatsoever. You can not add new files or directories and you can not change or delete existing files and directories. The only way to get rid of an immutable tree is to use the **s3qlrm** command (see below).

For example, to make the directory tree beneath the directory `2010-04-21` immutable, execute

```
s3qllock 2010-04-21
```

Immutability is a feature designed for backups. Traditionally, backups have been made on external tape drives. Once a backup was made, the tape drive was removed and locked somewhere in a shelf. This has the great advantage that the contents of the backup are now permanently fixed. Nothing (short of physical destruction) can change or delete files in the backup.

In contrast, when backing up into an online storage system like S3QL, all backups are available every time the file system is mounted. Nothing prevents a file in an old backup from being changed again later on. In the worst case, this may make your entire backup system worthless. Imagine that your system gets infected by a nasty virus that simply deletes all files it can find – if the virus is active while the backup file system is mounted, the virus will destroy all your old backups as well!

Even if the possibility of a malicious virus or trojan horse is excluded, being able to change a backup after it has been made is generally not a good idea. A common S3QL use case is to keep the file system mounted at all times and periodically create backups with **rsync -a**. This allows every user to recover her files from a backup without having to call the system administrator. However, this also allows every user to accidentally change or delete files *in* one of the old backups.

Making a backup immutable protects you against all these problems. Unless you happen to run into a virus that was specifically programmed to attack S3QL file systems, backups can be neither deleted nor changed after they have been made immutable.

8.4 Fast Recursive Removal

The `s3qlrm` command can be used to recursively delete files and directories on an S3QL file system. Although `s3qlrm` is faster than using e.g. `rm -r`, the main reason for its existence is that it allows you to delete immutable trees as well. The syntax is rather simple:

```
s3qlrm <directory>
```

Be warned that there is no additional confirmation. The directory will be removed entirely and immediately.

8.5 Runtime Configuration

The `s3qlctrl` can be used to control a mounted S3QL file system. Its syntax is

```
s3qlctrl [options] <action> <mountpoint> ...
```

`<mountpoint>` must be the location of a mounted S3QL file system. For a list of valid options, run `s3qlctrl --help`. `<action>` may be either of:

flushcache Flush file system cache. The command blocks until the cache has been flushed.

log Change log level.

cachesize Change file system cache size.

upload-meta Trigger a metadata upload.

CHAPTER

NINE

UNMOUNTING

To unmount an S3QL file system, use the command:

```
umount.s3ql [options] <mountpoint>
```

This will block until all data has been written to the backend.

Only the user who mounted the file system with **mount.s3ql** is able to unmount it again. If you are root and want to unmount an S3QL file system mounted by an ordinary user, you have to use the **fusermount -u** or **umount** command instead. Note that these commands do not block until all data has been uploaded, so if you use them instead of `umount.s3ql` then you should manually wait for the `mount.s3ql` process to terminate before shutting down the system.

The **umount.s3ql** command accepts the following options:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit
--lazy, -z	Lazy umount. Detaches the file system immediately, even if there are still open files. The data will be uploaded in the background once all open files have been closed.

If, for some reason, the `umount.sql` command does not work, the file system can also be unmounted with `fusermount -u -z`. Note that this command will return immediately and the file system may continue to upload data in the background for a while longer.

CHAPTER

TEN

CHECKING FOR ERRORS

If, for some reason, the filesystem has not been correctly unmounted, or if you suspect that there might be errors, you should run the `fsck.s3ql` utility. It has the following syntax:

```
fsck.s3ql [options] <storage url>
```

This command accepts the following options:

--log <target> Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `~/.s3ql/fsck.log`

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

--batch If user input is required, exit without prompting.

--force Force checking even if file system is marked clean.

CHAPTER

ELEVEN

STORING AUTHENTICATION INFORMATION

Normally, S3QL reads username and password for the backend as well as an encryption passphrase for the file system from the terminal. Most commands also accept an `--authfile` parameter that can be used to read this information from a file instead.

The authentication file consists of sections, led by a `[section]` header and followed by `name: value` entries. The section headers themselves are not used by S3QL but have to be unique within the file.

In each section, the following entries can be defined:

storage-url Specifies the storage url to which this section applies. If a storage url starts with the value of this entry, the section is considered applicable.

backend-login Specifies the username to use for authentication with the backend.

backend-password Specifies the password to use for authentication with the backend.

fs-passphrase Specifies the passphrase to use to decrypt the file system (if it is encrypted).

When reading the authentication file, S3QL considers every applicable section in order and uses the last value that it found for each entry. For example, consider the following authentication file:

```
[s3]
storage-url: s3://
backend-login: joe
backend-password: notquitesecret

[fs1]
storage-url: s3://joes-first-bucket
fs-passphrase: neitheristhis

[fs2]
storage-url: s3://joes-second-bucket
fs-passphrase: swordfish

[fs3]
storage-url: s3://joes-second-bucket/with-prefix
backend-login: bill
backend-password: bi23ll
fs-passphrase: ll23bi
```

With this authentication file, S3QL would try to log in as "joe" whenever the s3 backend is used, except when accessing a storage url that begins with "s3://joes-second-bucket/with-prefix". In that case, the last section becomes active and

S3QL would use the “bill” credentials. Furthermore, file system encryption passphrases will be used for storage urls that start with “s3://joes-first-bucket” or “s3://joes-second-bucket”.

The authentication file is parsed by the Python ConfigParser module.

CHAPTER

TWELVE

CONTRIBUTED PROGRAMS

S3QL comes with a few contributed programs that are not part of the core distribution (and are therefore not installed automatically by default), but which may nevertheless be useful. These programs are in the `contrib` directory of the source distribution or in `/usr/share/doc/s3ql/contrib` if you installed S3QL from a package.

12.1 benchmark.py

This program measures S3QL write performance, uplink bandwidth and compression speed to determine the limiting factor. It also gives recommendation for compression algorithm and number of upload threads to achieve maximum performance.

12.2 s3_copy.py

This program physically duplicates Amazon S3 bucket. It can be used to migrate buckets to a different storage region or storage class (standard or reduced redundancy).

12.3 pcp.py

`pcp.py` is a wrapper program that starts several rsync processes to copy directory trees in parallel. This is important because transferring files in parallel significantly enhances performance when copying data from an S3QL file system (see *Improving copy performance* for details).

To recursively copy the directory `/mnt/home-backup` into `/home/joe` using 8 parallel processes and preserving permissions, you would execute

```
pcp.py -a --processes=8 /mnt/home-backup/ /home/joe
```

12.4 s3_backup.sh

This is an example script that demonstrates how to set up a simple but powerful backup solution using S3QL and rsync.

The `s3_backup.sh` script automates the following steps:

1. Mount the file system

2. Replicate the previous backup with *s3qlcp*
3. Update the new copy with the data from the backup source using rsync
4. Make the new backup immutable with *s3qllock*
5. Delete old backups that are no longer needed
6. Unmount the file system

The backups are stored in directories of the form `YYYY-MM-DD_HH:mm:SS` and the expire_backups.py command is used to delete old backups.

12.5 expire_backups.py

expire_backups.py is a program to intelligently remove old backups that are no longer needed.

To define what backups you want to keep for how long, you define a number of *age ranges*. **expire_backups** ensures that you will have at least one backup in each age range at all times. It will keep exactly as many backups as are required for that and delete any backups that become redundant.

Age ranges are specified by giving a list of range boundaries in terms of backup cycles. Every time you create a new backup, the existing backups age by one cycle.

Example: when **expire_backups** is called with the age range definition `1 3 7 14 31`, it will guarantee that you always have the following backups available:

1. A backup that is 0 to 1 cycles old (i.e, the most recent backup)
2. A backup that is 1 to 3 cycles old
3. A backup that is 3 to 7 cycles old
4. A backup that is 7 to 14 cycles old
5. A backup that is 14 to 31 cycles old

Note: If you do backups in fixed intervals, then one cycle will be equivalent to the backup interval. The advantage of specifying the age ranges in terms of backup cycles rather than days or weeks is that it allows you to gracefully handle irregular backup intervals. Imagine that for some reason you do not turn on your computer for one month. Now all your backups are at least a month old, and if you had specified the above backup strategy in terms of absolute ages, they would all be deleted! Specifying age ranges in terms of backup cycles avoids these sort of problems.

---

expire_backups usage is simple. It requires backups to have names of the forms `year-month-day_hour:minute:seconds` (`YYYY-MM-DD_HH:mm:ss`) and works on all backups in the current directory. So for the above backup strategy, the correct invocation would be:

```
expire_backups.py 1 3 7 14 31
```

When storing your backups on an S3QL file system, you probably want to specify the `--use-s3qlrm` option as well. This tells **expire_backups** to use the *s3qlrm* command to delete directories.

expire_backups uses a "state file" to keep track which backups are how many cycles old (since this cannot be inferred from the dates contained in the directory names). The standard name for this state file is `.expire_backups.dat`. If this file gets damaged or deleted, **expire_backups** no longer knows the ages of the backups and refuses to work. In this case you can use the `--reconstruct-state` option to try to reconstruct the state from the backup dates. However, the accuracy of this reconstruction depends strongly on how rigorous you have been with making backups (it is only completely correct if the time between subsequent backups has always been exactly the same), so it's generally a good idea not to tamper with the state file.

For a full list of available options, run **expire_backups.py –help**.

12.6 s3ql_upstart.conf

`s3ql_upstart.conf` is an example upstart job definition file. It defines a job that automatically mounts an S3QL file system on system start, and properly unmounts it when the system is shut down.

CHAPTER

THIRTEEN

TIPS & TRICKS

13.1 SSH Backend

By combining S3QL's local backend with sshfs, it is possible to store an S3QL file system on arbitrary SSH servers: first mount the remote target directory into the local filesystem,

```
sshfs user@my.server.com:/mnt/s3ql /mnt/sshfs
```

and then give the mountpoint to S3QL as a local destination:

```
mount.s3ql local:///mnt/sshfs/myfsdata /mnt/s3ql
```

13.2 Permanently mounted backup file system

If you use S3QL as a backup file system, it can be useful to mount the file system permanently (rather than just mounting it for a backup and unmounting it afterwards). Especially if your file system becomes large, this saves you long mount- and unmount times if you only want to restore a single file.

If you decide to do so, you should make sure to

- Use *s3qllock* to ensure that backups are immutable after they have been made.
- Call *s3qlctrl upload-meta* right after a every backup to make sure that the newest metadata is stored safely (if you do backups often enough, this may also allow you to set the `--metadata-upload-interval` option of **mount.s3ql** to zero).

13.3 Improving copy performance

Note: The following applies only when copying data **from** an S3QL file system, **not** when copying data **to** an S3QL file system.

If you want to copy a lot of smaller files *from* an S3QL file system (e.g. for a system restore) you will probably notice that the performance is rather bad.

The reason for this is intrinsic to the way S3QL works. Whenever you read a file, S3QL first has to retrieve this file over the network from the backend. This takes a minimum amount of time (the network latency), no matter how big or small the file is. So when you copy lots of small files, 99% of the time is actually spend waiting for network data.

Theoretically, this problem is easy to solve: you just have to copy several files at the same time. In practice, however, almost all unix utilities (`cp`, `rsync`, `tar` and friends) insist on copying data one file at a time. This makes a lot of sense when copying data on the local hard disk, but in case of S3QL this is really unfortunate.

The best workaround that has been found so far is to copy files by starting several rsync processes at once and use exclusion rules to make sure that they work on different sets of files.

For example, the following script will start 3 rsync instances. The first instance handles all filenames starting with a-f, the second the filenames from g-l and the third covers the rest. The `+ */` rule ensures that every instance looks into all directories.

```
#!/bin/bash

RSYNC_ARGS="-aHv /mnt/s3ql/ /home/restore/"

rsync -f "+ */" -f "-! [a-f]*" $RSYNC_ARGS &
rsync -f "+ */" -f "-! [g-l]*" $RSYNC_ARGS &
rsync -f "+ */" -f "- [a-l]*" $RSYNC_ARGS &

wait
```

The optimum number of parallel processes depends on your network connection and the size of the files that you want to transfer. However, starting about 10 processes seems to be a good compromise that increases performance dramatically in almost all situations.

S3QL comes with a script named `pcp.py` in the `contrib` directory that can be used to transfer files in parallel without having to write an explicit script first. See the description of *pcp.py* for details.

CHAPTER

FOURTEEN

KNOWN ISSUES

- S3QL does not support Access Control Lists (ACLs). This is due to a bug in the FUSE library and will therefore hopefully be fixed at some point. See issue 385 for more details.
- S3QL does not verify TLS/SSL server certificates and is thus vulnerable to man-in-the-middle attacks. See issue 267 for more details.
- S3QL is rather slow when an application tries to write data in unreasonably small chunks. If a 1 MiB file is copied in chunks of 1 KB, this will take more than 10 times as long as when it's copied with the (recommended) chunk size of 128 KiB.

  This is a limitation of the FUSE library (which does not yet support write caching) which will hopefully be addressed in some future FUSE version.

  Most applications, including e.g. GNU `cp` and `rsync`, use reasonably large buffers and are therefore not affected by this problem and perform very efficient on S3QL file systems.

  However, if you encounter unexpectedly slow performance with a specific program, this might be due to the program using very small write buffers. Although this is not really a bug in the program, it might be worth to ask the program's authors for help.
- S3QL always updates file and directory access times as if the `relatime` mount option has been specified: the access time ("atime") is only updated if it is currently earlier than either the status change time ("ctime") or modification time ("mtime").
- S3QL directories always have an `st_nlink` value of 1. This may confuse programs that rely on directories having `st_nlink` values of *(2 + number of sub directories)*.

  Note that this is not a bug in S3QL. Including sub directories in the `st_nlink` value is a Unix convention, but by no means a requirement. If an application blindly relies on this convention being followed, then this is a bug in the application.

  A prominent example are early versions of GNU find, which required the `--noleaf` option to work correctly on S3QL file systems. This bug has already been fixed in recent find versions.
- The `umount` and `fusermount -u` commands will *not* block until all data has been uploaded to the backend. (this is a FUSE limitation that will hopefully be removed in the future, see issue 159). If you use either command to unmount an S3QL file system, you have to take care to explicitly wait for the `mount.s3ql` process to terminate before you shut down or restart the system. Therefore it is generally not a good idea to mount an S3QL file system in `/etc/fstab` (you should use a dedicated init script instead).
- S3QL relies on the backends not to run out of space. This is a given for big storage providers like Amazon S3 or Google Storage, but you may stumble upon this if you use your own server or smaller providers.

  If there is no space left in the backend, attempts to write more data into the S3QL file system will fail and the file system will be in an inconsistent state and require a file system check (and you should make sure to make space available in the backend before running the check).

Unfortunately, there is no way to handle insufficient space in the backend without leaving the file system inconsistent. Since S3QL first writes data into the cache, it can no longer return an error when it later turns out that the cache can not be committed to the backend.

CHAPTER

FIFTEEN

MANPAGES

The man pages are installed with S3QL on your system and can be viewed with the **man** command. For reference, they are also included here in the User's Guide.

15.1 The mkfs.s3ql command

15.1.1 Synopsis

```
mkfs.s3ql [options] <storage url>
```

15.1.2 Description

The **mkfs.s3ql** command creates a new file system in the location specified by *storage url*. The storage url depends on the backend that is used. The S3QL User's Guide should be consulted for a description of the available backends.

Unless you have specified the `--plain` option, `mkfs.s3ql` will ask you to enter an encryption password. This password will *not* be read from an authentication file specified with the `--authfile` option to prevent accidental creation of an encrypted file system.

15.1.3 Options

The **mkfs.s3ql** command accepts the following options.

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

-L <name> Filesystem label

--max-obj-size <size> Maximum size of storage objects in KiB. Files bigger than this will be spread over multiple objects in the storage backend. Default: 10240 KiB.

--plain Create unencrypted file system.

--force Overwrite any existing data.

15.1.4 Exit Status

mkfs.s3ql returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.1.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.2 The s3qladm command

15.2.1 Synopsis

`s3qladm` *[options]* `<action> <storage url>`

where `action` may be either of **passphrase**, **upgrade**, **delete** or **download-metadata**.

15.2.2 Description

The **s3qladm** command performs various operations on *unmounted* S3QL file systems. The file system *must not be mounted* when using **s3qladm** or things will go wrong badly.

The storage url depends on the backend that is used. The S3QL User's Guide should be consulted for a description of the available backends.

15.2.3 Options

The **s3qladm** command accepts the following options.

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--log <target> Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `none`

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--ssl	Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.
--cachedir <path>	Store cached data in this directory (default: `~/.s3ql`)
--version	just print program version and exit

Hint: run `s3qladm <action> --help` to get help on the additional arguments that the different actions take.

15.2.4 Actions

The following actions may be specified:

passphrase Changes the encryption passphrase of the file system.

upgrade Upgrade the file system to the newest revision.

delete Delete the file system with all the stored data.

download-metadata Interactively download backups of the file system metadata.

15.2.5 Exit Status

s3qladm returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.2.6 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.3 The mount.s3ql command

15.3.1 Synopsis

`mount.s3ql` *[options]* `<storage url> <mount point>`

15.3.2 Description

The **mount.s3ql** command mounts the S3QL file system stored in *storage url* in the directory *mount point*. The storage url depends on the backend that is used. The S3QL User's Guide should be consulted for a description of the available backends.

15.3.3 Options

The **mount.s3ql** command accepts the following options.

--log <target>	Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `~/.s3ql/mount.log`

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

--cachesize <size> Cache size in KiB (default: 102400 (100 MiB)). Should be at least 10 times the maximum object size of the filesystem, otherwise an object may be retrieved and written several times during a single write() or read() operation.

--max-cache-entries <num> Maximum number of entries in cache (default: 768). Each cache entry requires one file descriptor, so if you increase this number you have to make sure that your process file descriptor limit (as set with `ulimit -n`) is high enough (at least the number of cache entries + 100).

--allow-other Normally, only the user who called `mount.s3ql` can access the mount point. This user then also has full access to it, independent of individual file permissions. If the `--allow-other` option is specified, other users can access the mount point as well and individual file permissions are taken into account for all users.

--allow-root Like `--allow-other`, but restrict access to the mounting user and the root user.

--fg Do not daemonize, stay in foreground

--single Run in single threaded mode. If you don't understand this, then you don't need it.

--upstart Stay in foreground and raise SIGSTOP once mountpoint is up.

--profile Create profiling information. If you don't understand this, then you don't need it.

--compress <name> Compression algorithm to use when storing new data. Allowed values: `lzma`, `bzip2`, `zlib`, none. (default: `lzma`)

--metadata-upload-interval <seconds> Interval in seconds between complete metadata uploads. Set to 0 to disable. Default: 24h.

--threads <no> Number of parallel upload threads to use (default: auto).

--nfs Enable some optimizations for exporting the file system over NFS. (default: False)

15.3.4 Exit Status

mount.s3ql returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.3.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.4 The s3qlstat command

15.4.1 Synopsis

`s3qlstat` *[options]* `<mountpoint>`

15.4.2 Description

The **s3qlstat** command prints statistics about the S3QL file system mounted at `mountpoint`.

s3qlstat can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

15.4.3 Options

The **s3qlstat** command accepts the following options:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit

15.4.4 Exit Status

s3qlstat returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.4.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.5 The s3qlctrl command

15.5.1 Synopsis

`s3qlctrl` *[options]* `<action>` `<mountpoint>` `...`

where `action` may be either of **flushcache**, **upload-meta**, **cachesize** or **log-metadata**.

15.5.2 Description

The **s3qlctrl** command performs various actions on the S3QL file system mounted in `mountpoint`.

s3qlctrl can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

The following actions may be specified:

flushcache Uploads all changed file data to the backend.

upload-meta Upload metadata to the backend. All file system operations will block while a snapshot of the metadata is prepared for upload.

cachesize Changes the cache size of the file system. This action requires an additional argument that specifies the new cache size in KiB, so the complete command line is:

```
s3qlctrl [options] cachesize <mountpoint> <new-cache-size>
```

log Change the amount of information that is logged into `~/.s3ql/mount.log` file. The complete syntax is:

```
s3qlctrl [options] log <mountpoint> <level> [<module> [<module> ...]]
```

here `level` is the desired new log level and may be either of *debug*, *info* or *warn*. One or more `module` may only be specified with the *debug* level and allow to restrict the debug output to just the listed modules.

15.5.3 Options

The **s3qlctrl** command also accepts the following options, no matter what specific action is being invoked:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit

Hint: run `s3qlctrl <action> --help` to get help on the additional arguments that the different actions take.

15.5.4 Exit Status

s3qlctrl returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.5.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.6 The s3qlcp command

15.6.1 Synopsis

```
s3qlcp [options] <source-dir> <dest-dir>
```

15.6.2 Description

The **s3qlcp** command duplicates the directory tree `source-dir` into `dest-dir` without physically copying the file contents. Both source and destination must lie inside the same S3QL file system.

The replication will not take any additional space. Only if one of directories is modified later on, the modified data will take additional storage space.

`s3qlcp` can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

Note that:

- After the replication, both source and target directory will still be completely ordinary directories. You can regard `<src>` as a snapshot of `<target>` or vice versa. However, the most common usage of `s3qlcp` is to regularly duplicate the same source directory, say `documents`, to different target directories. For a e.g. monthly replication, the target directories would typically be named something like `documents_January` for the replication in January, `documents_February` for the replication in February etc. In this case it is clear that the target directories should be regarded as snapshots of the source directory.
- Exactly the same effect could be achieved by an ordinary copy program like `cp -a`. However, this procedure would be orders of magnitude slower, because `cp` would have to read every file completely (so that S3QL had to fetch all the data over the network from the backend) before writing them into the destination folder.

Snapshotting vs Hardlinking

Snapshot support in S3QL is inspired by the hardlinking feature that is offered by programs like rsync or storeBackup. These programs can create a hardlink instead of copying a file if an identical file already exists in the backup. However, using hardlinks has two large disadvantages:

- backups and restores always have to be made with a special program that takes care of the hardlinking. The backup must not be touched by any other programs (they may make changes that inadvertently affect other hardlinked files)
- special care needs to be taken to handle files which are already hardlinked (the restore program needs to know that the hardlink was not just introduced by the backup program to safe space)

S3QL snapshots do not have these problems, and they can be used with any backup program.

15.6.3 Options

The **s3qlcp** command accepts the following options:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit

15.6.4 Exit Status

s3qlcp returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.6.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.7 The s3qlrm command

15.7.1 Synopsis

`s3qlrm` *[options]* `<directory>`

15.7.2 Description

The **s3qlrm** command recursively deletes files and directories on an S3QL file system. Although **s3qlrm** is faster than using e.g. **rm -r'**, the main reason for its existence is that it allows you to delete immutable trees (which can be created with **s3qllock**) as well.

Be warned that there is no additional confirmation. The directory will be removed entirely and immediately.

s3qlrm can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

15.7.3 Options

The **s3qlrm** command accepts the following options:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit

15.7.4 Exit Status

s3qlrm returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.7.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.8 The s3qllock command

15.8.1 Synopsis

```
s3qllock [options] <directory>
```

15.8.2 Description

The **s3qllock** command makes a directory tree in an S3QL file system immutable. Immutable trees can no longer be changed in any way whatsoever. You can not add new files or directories and you can not change or delete existing files and directories. The only way to get rid of an immutable tree is to use the **s3qlrm** command.

s3qllock can only be called by the user that mounted the file system and (if the file system was mounted with `--allow-other` or `--allow-root`) the root user. This limitation might be removed in the future (see issue 155).

15.8.3 Rationale

Immutability is a feature designed for backups. Traditionally, backups have been made on external tape drives. Once a backup was made, the tape drive was removed and locked somewhere in a shelf. This has the great advantage that the contents of the backup are now permanently fixed. Nothing (short of physical destruction) can change or delete files in the backup.

In contrast, when backing up into an online storage system like S3QL, all backups are available every time the file system is mounted. Nothing prevents a file in an old backup from being changed again later on. In the worst case, this may make your entire backup system worthless. Imagine that your system gets infected by a nasty virus that simply deletes all files it can find – if the virus is active while the backup file system is mounted, the virus will destroy all your old backups as well!

Even if the possibility of a malicious virus or trojan horse is excluded, being able to change a backup after it has been made is generally not a good idea. A common S3QL use case is to keep the file system mounted at all times and periodically create backups with **rsync -a**. This allows every user to recover her files from a backup without having to call the system administrator. However, this also allows every user to accidentally change or delete files *in* one of the old backups.

Making a backup immutable protects you against all these problems. Unless you happen to run into a virus that was specifically programmed to attack S3QL file systems, backups can be neither deleted nor changed after they have been made immutable.

15.8.4 Options

The **s3qllock** command accepts the following options:

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit

15.8.5 Exit Status

s3qllock returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.8.6 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.9 The umount.s3ql command

15.9.1 Synopsis

```
umount.s3ql [options] <mount point>
```

15.9.2 Description

The **umount.s3ql** command unmounts the S3QL file system mounted in the directory *mount point* and blocks until all data has been uploaded to the storage backend.

Only the user who mounted the file system with **mount.s3ql** is able to unmount it with **umount.s3ql**. If you are root and want to unmount an S3QL file system mounted by an ordinary user, you have to use the **fusermount -u** or **umount** command instead. Note that these commands do not block until all data has been uploaded, so if you use them instead of **umount.s3ql** then you should manually wait for the **mount.s3ql** process to terminate before shutting down the system.

15.9.3 Options

The **umount.s3ql** command accepts the following options.

--debug	activate debugging output
--quiet	be really quiet
--version	just print program version and exit
--lazy, -z	Lazy umount. Detaches the file system immediately, even if there are still open files. The data will be uploaded in the background once all open files have been closed.

15.9.4 Exit Status

umount.s3ql returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.9.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.10 The fsck.s3ql command

15.10.1 Synopsis

```
fsck.s3ql [options] <storage url>
```

15.10.2 Description

The **mkfs.s3ql** command checks the new file system in the location specified by *storage url* for errors and attempts to repair any problems. The storage url depends on the backend that is used. The S3QL User's Guide should be consulted for a description of the available backends.

15.10.3 Options

The **mkfs.s3ql** command accepts the following options.

--log <target> Write logging info into this file. File will be rotated when it reaches 1 MiB, and at most 5 old log files will be kept. Specify `none` to disable logging. Default: `~/.s3ql/fsck.log`

--cachedir <path> Store cached data in this directory (default: `~/.s3ql`)

--authfile <path> Read authentication credentials from this file (default: `~/.s3ql/authinfo2`)

--debug <module> activate debugging output from <module>. Use `all` to get debug messages from all modules. This option can be specified multiple times.

--quiet be really quiet

--ssl Always use SSL connections when connecting to remote servers. For backends that allow only encrypted connections, S3QL uses SSL automatically, even if this option is not set.

--version just print program version and exit

--batch If user input is required, exit without prompting.

--force Force checking even if file system is marked clean.

15.10.4 Exit Status

mkfs.s3ql returns exit code 0 if the operation succeeded and 1 if some error occurred.

15.10.5 See Also

The S3QL homepage is at http://code.google.com/p/s3ql/.

The full S3QL documentation should also be installed somewhere on your system, common locations are `/usr/share/doc/s3ql` or `/usr/local/doc/s3ql`.

15.11 The pcp command

15.11.1 Synopsis

```
pcp [options] <source> [<source> ...] <destination>
```

15.11.2 Description

The **pcp** command is a is a wrapper that starts several **sync** processes to copy directory trees in parallel. This is allows much better copying performance on file system that have relatively high latency when retrieving individual files like S3QL.

Note: Using this program only improves performance when copying *from* an S3QL file system. When copying *to* an S3QL file system, using **pcp** is more likely to *decrease* performance.

15.11.3 Options

The **pcp** command accepts the following options:

--quiet	be really quiet
--debug	activate debugging output
--version	just print program version and exit
-a	Pass -aHAX option to rsync.
--processes <no>	Number of rsync processes to use (default: 10).

15.11.4 Exit Status

pcp returns exit code 0 if the operation succeeded and 1 if some error occured.

15.11.5 See Also

pcp is shipped as part of S3QL, http://code.google.com/p/s3ql/.

15.12 The expire_backups command

15.12.1 Synopsis

```
expire_backups [options] <age> [<age> ...]
```

15.12.2 Description

The **expire_backups** command intelligently remove old backups that are no longer needed.

To define what backups you want to keep for how long, you define a number of *age ranges*. **expire_backups** ensures that you will have at least one backup in each age range at all times. It will keep exactly as many backups as are required for that and delete any backups that become redundant.

Age ranges are specified by giving a list of range boundaries in terms of backup cycles. Every time you create a new backup, the existing backups age by one cycle.

Example: when **expire_backups** is called with the age range definition `1 3 7 14 31`, it will guarantee that you always have the following backups available:

1. A backup that is 0 to 1 cycles old (i.e, the most recent backup)
2. A backup that is 1 to 3 cycles old
3. A backup that is 3 to 7 cycles old
4. A backup that is 7 to 14 cycles old
5. A backup that is 14 to 31 cycles old

Note: If you do backups in fixed intervals, then one cycle will be equivalent to the backup interval. The advantage of specifying the age ranges in terms of backup cycles rather than days or weeks is that it allows you to gracefully handle irregular backup intervals. Imagine that for some reason you do not turn on your computer for one month. Now all your backups are at least a month old, and if you had specified the above backup strategy in terms of absolute ages, they would all be deleted! Specifying age ranges in terms of backup cycles avoids these sort of problems.

---

expire_backups usage is simple. It requires backups to have names of the forms `year-month-day_hour:minute:seconds` (`YYYY-MM-DD_HH:mm:ss`) and works on all backups in the current directory. So for the above backup strategy, the correct invocation would be:

```
expire_backups.py 1 3 7 14 31
```

When storing your backups on an S3QL file system, you probably want to specify the `--use-s3qlrm` option as well. This tells **expire_backups** to use the *s3qlrm* command to delete directories.

expire_backups uses a "state file" to keep track which backups are how many cycles old (since this cannot be inferred from the dates contained in the directory names). The standard name for this state file is `.expire_backups.dat`. If this file gets damaged or deleted, **expire_backups** no longer knows the ages of the backups and refuses to work. In this case you can use the `--reconstruct-state` option to try to reconstruct the state from the backup dates. However, the accuracy of this reconstruction depends strongly on how rigorous you have been with making backups (it is only completely correct if the time between subsequent backups has always been exactly the same), so it's generally a good idea not to tamper with the state file.

15.12.3 Options

The **expire_backups** command accepts the following options:

--quiet	be really quiet
--debug	activate debugging output
--version	just print program version and exit
--state <file>	File to save state information in (default: ".expire_backups.dat")
-n	Dry run. Just show which backups would be deleted.
--reconstruct-state	Try to reconstruct a missing state file from backup dates.
--use-s3qlrm	Use `s3qlrm` command to delete backups.

15.12.4 Exit Status

expire_backups returns exit code 0 if the operation succeeded and 1 if some error occured.

15.12.5 See Also

expire_backups is shipped as part of S3QL, http://code.google.com/p/s3ql/.

CHAPTER

SIXTEEN

FURTHER RESOURCES / GETTING HELP

If you have questions or problems with S3QL that you weren't able to resolve with this manual, you might want to consider the following other resources:

- The S3QL Wiki
- The S3QL FAQ
- The S3QL Mailing List. You can subscribe by sending a mail to s3ql+subscribe@googlegroups.com.

Please report any bugs you may encounter in the Issue Tracker.

CHAPTER

SEVENTEEN

IMPLEMENTATION DETAILS

This section provides some background information on how S3QL works internally. Reading this section is not necessary to use S3QL.

17.1 Metadata Storage

Like most unix filesystems, S3QL has a concept of inodes.

The contents of directory inodes (aka the names and inodes of the files and sub directories contained in a directory) are stored directly in a [http://www.sqlite.org SQLite] database. This database is stored in a special S3 object that is downloaded when the file system is mounted and uploaded periodically in the background and when the file system is unmounted. This has two implications:

1. The entire file system tree can be read from the database. Fetching or storing S3 objects is only required to access the contents of files (or, more precisely, inodes). This makes most file system operations very fast because no data has to be send over the network.
2. An S3QL filesystem can only be mounted at one position at a time, otherwise changes made in one mountpoint will invariably be overwritten when the second mount point is unmounted.

Sockets, FIFOs and character devices do not need any additional storage, all information about them is contained in the database.

17.2 Data Storage

The contents of file inodes are split into individual blocks. The maximum size of a block is specified when the file system is created and cannot be changed afterwards. Every block is stored as an individual object in the backend, and the mapping from inodes to blocks and from blocks to objects is stored in the database.

While the file system is mounted, blocks are cached locally.

Blocks can also be compressed and encrypted before they are stored in S3.

If some files have blocks with identical contents, the blocks will be stored in the same backend object (i.e., the data is only stored once).

17.3 Data De-Duplication

Instead of uploading every block, S3QL first computes a checksum (a SHA256 hash, for those who are interested) to check if an identical blocks has already been stored in an backend object. If that is the case, the new block will be

linked to the existing object instead of being uploaded.

This procedure is invisible for the user and the contents of the block can still be changed. If several blocks share a backend object and one of the blocks is changed, the changed block is automatically stored in a new object (so that the contents of the other block remain unchanged).

17.4 Caching

When an application tries to read or write from a file, S3QL determines the block that contains the required part of the file and retrieves it from the backend or creates it if it does not yet exist. The block is then held in the cache directory. It is committed to S3 when it has not been accessed for more than 10 seconds. Blocks are removed from the cache only when the maximum cache size is reached.

When the file system is unmounted, all modified blocks are committed to the backend and the cache is cleaned.

17.5 Eventual Consistency Handling

S3QL has to take into account that changes in objects do not propagate immediately in all backends. For example, when an Amazon S3 object is uploaded and immediately downloaded again, the downloaded data might not yet reflect the changes done in the upload (see also http://developer.amazonwebservices.com/connect/message.jspa?messageID=38538)

For the data blocks this is not a problem because a data blocks always get a new object ID when they are updated.

For the metadata however, S3QL has to make sure that it always downloads the most recent copy of the database when mounting the file system.

To that end, metadata versions are numbered, and the most recent version number is stored as part of the object id of a very small "marker" object. When S3QL has downloaded the metadata it checks the version number against the marker object and, if the two do not agree, waits for the most recent metadata to become available. Once the current metadata is available, the version number is increased and the marker object updated.

17.6 Encryption

When the file system is created, `mkfs.s3ql` generates a 256 bit master key by reading from `/dev/random`. The master key is encrypted with the passphrase that is entered by the user, and then stored with the rest of the file system data. Since the passphrase is only used to access the master key (which is used to encrypt the actual file system data), the passphrase can easily be changed.

Data is encrypted with a new session key for each object and each upload. The session key is generated by appending a nonce to the master key and then calculating the SHA256 hash. The nonce is generated by concatenating the object id and the current UTC time as a 32 bit float. The precision of the time is given by the [http://docs.python.org/library/time.html#time.time Python `time()` function] and usually at least 1 millisecond. The SHA256 implementation is included in the Python standard library.

Once the session key has been calculated, a SHA256 HMAC is calculated over the data that is to be uploaded. Afterwards, the data is compressed with the LZMA, [http://en.wikipedia.org/wiki/Bz2 Bz2 algorithm] or LZ and the HMAC inserted at the beginning. Both HMAC and compressed data are then encrypted using 256 bit AES in CTR mode. The AES-CTR implementation is provided by the [http://cryptopp.com/ Crypto++] library. Finally, the nonce is inserted in front of the encrypted data and HMAC, and the packet is send to the backend as a new S3 object.