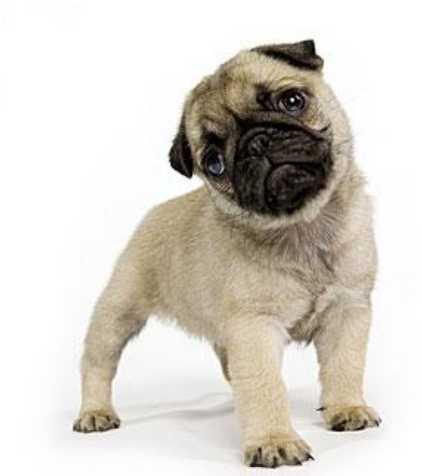

(Photo: deanpalmer.ca)

Pug System Overview

Version: 1.6

March 27, 2025

Contents

Software License

This software is licensed as "shareware" and may be used free of charge. If you find the software useful, a donation would be appreciated for continued use.

Pug utilizes AES Crypt for encryption, which requires a separate software license.

This software is provided "as is" and without any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The author shall not be held liable for any damages resulting from the use of this software, either directly or indirectly, including, but not limited to, loss of data or data being rendered inaccurate.

Copyright and Trademark Notices

1 What is Pug?

Pug™ is cloud-based backup software designed to run on Linux or via a container. Inspiration for Pug came from a general dislike for handling daily or weekly backup chores, combined with the fact that storage on any external physical medium means that one needs an external storage facility to avoid loss due to fire or other natural disasters.

Many existing cloud-backup solutions work by creating a compressed archive of data to be backed up and then pushing that huge file into the cloud. This consumes way more space than is necessary, as files are redundantly stored, and it takes far too long to backup data into the cloud.

Pug takes a different, more granular approach to the backup problem. With Pug, one defines a set of "locations" (i.e., directories) to archive. Periodically, Pug's "discover" process will scan those locations looking for new or modified files and will note them in the local Pug database. The Pug "archiver" process also run periodically to archive any new or modified files to cloud storage, compressing and encrypting each file before transmitting the file to the cloud. Archived files that are identical are only stored in the cloud one time to avoid wasted space and bandwidth. Thus, from day-to-day, the only files uploaded to cloud storage are new or modified files that are not already stored in the cloud. At any given point in time after the archiver completes and files have existed long enough to satisfy the archival delay period, all files that exist on your local machine also exist in the cloud.

Pug also allows one to maintain historical copies of files for a specified period. Files that are deleted from the local machine remain in cloud storage until Pug is told to expunge the files. As the Pug administrator, you have to ability to see every file stored in cloud storage and to retrieve any version of the file using the "pget" command.

In the event of a system crash and all local copies of files are lost, one can recover files from Pug using the "precover" command. That command will recover the most recent version of all files that existed on or after the date specified on the command-line.

In short, Pug is an incremental, secure, and efficient cloud archiving solution that eliminates the need for traditional, tedious backup methods employed in the past.

2 Technology Employed

Pug is written entirely in the popular scripting language Perl, meaning that it should be easily ported to any Linux machine without any effort. It was originally written using Perl 5.14, so anything newer should be compatible (the latest tested version is 5.34). There was no intent or desire to use "cutting edge" Perl features, as the primary objective was to ensure portability of code and ease of system recovery in the event of a failure.

Pug relies on MySQL or MariaDB for data storage. While either may be used, this document refers only to MySQL for brevity (except in the container section where MariaDB is used as an example). MySQL was selected for performance reasons and for the fact that tools like phpMyAdmin and MySQL Workbench make it simple to administer.

Pug utilizes Amazon S3 (https://aws.amazon.com/s3/) for file storage. Amazon offers a robust storage service that virtually guarantees no loss of data.

Pug uses AES Crypt™ for file encryption. This software is available from https://www.aescrypt.com/ and must be installed, as all files are encrypted with that tool before uploading to the cloud. While cloud storage providers offer secure transmission (using HTTPS) and cloud-side encryption options, relying on a cloud storage provider to provide encryption means that, should someone gain access to the cloud account, they could access to file stored there. It is for that reason that files are encrypted before transmission to the cloud using keys you create and control, thus helping to keep your data secure from unauthorized access.

3 System Overview

Pug was selected as the name for this software, because there seems to be consensus among dog owners that pugs are some of the laziest dogs in the world. It was laziness to consistently deal with off-site backups that inspired the creation of Pug. Also, Pug implements a somewhat lazy approach collecting files, archiving them, and is even lazier about getting rid of them once archived. These characteristics about Pug are good, as the result is a very efficient, smooth, transparent, and current archival of all data.

Pug is composed of the Pug database (which is document in section 8) and a set of programs that look for files to be archived, archive files, retrieve files from storage, and expunge files from storage. The software installation directory is structured as shown below and each file is briefly explained.

Directory	Tool	Description
/		
	Dockerfile	A Docker file that can be used to build a container for running Pug.
	LICENSE.md	A file containing a copy of the license, copyright, and trademark notices.
	README.md	A file containing references to documentation, performance considerations, version history, or other information for getting started.
bin/		
	pget	This is a tool that allows the administrator to get specific files from cloud storage, including previous versions of specific files.
	pharmony	This command will perform a consistency check between cloud storage and the local database to ensure that all files in cloud storage are supposed to be there and that all files in the database are in cloud storage. This command should be used sparingly and only when Pug services are not running. It is a very useful tool for identifying inconsistencies after performing a system recovery or repairing a damaged database.
	plocations	This tool will display the current set of defined “locations” that are to be archived, along with the current scanning and archival status.

	pls	This will provide a listing of all files currently archived by Pug, including multiple versions of archived files (if any).
	precover	This command will allow an administrator to recover all files for a given “location” or a sub-set of those files. This command always retrieves the most recent version of a file from cloud storage.
	ptime	This command will display the current Unix time (useful for use with precover) and optionally show the time some number of days prior
sbin/		
	pug_archiver	This program will archive all files marked in the local database by the discover process as new or potentially new files. The archiver is smart enough to recognize that a given file has already been archived and to simply refer to that archived file, versus uploading it again.
	pug_check	This script is called by several other programs to check to make sure that it is safe to perform discovery and archival functions. If this script returns a non-zero status code, the archiver, discover, and housekeeper processes will not run. This is useful to prevent Pug from running when file systems are not mounted or the system time is not synchronized, as examples. This script is intended to be modified by the system administrator and it should be observed that the script presently requires ntpd to be running and the time synchronized. If you do not want to force that requirement, comment out that line of code in the installation package. This file should be in the PUG_SCRIPTS path set in pug_env.
	pug_db_archiver	This program will archive the local Pug database in cloud storage along with other archived files. In the event of a total system failure, the administrator may retrieve the pug database from cloud storage manually, restore it, and then use the precover command to restore all archived data.
	pug_discover	This program will scan all locations in the database for which scanning is active and look for new or modified files, inserting filenames into the database upon which the archiver will act.
	pug_env	This is a shell script that sets environment variables for the benefit of other Pug scripts. One may move this file to any location, but it must be read before calling other scripts. This is where configuration should be done.
	pug_housekeeper	This program should be run periodically to perform various housekeeping chores, including expunging files from cloud storage that are no longer needed. This script also performs consistency checks and takes steps to ensure that any errors found in the database are corrected on the next run of the discover and archiver processes.
	pug_requires	This script exists only to provide verification that all required Perl modules are installed on the system. If any errors are produced when this script is executed, it means some module is missing and must be installed for Pug to operate properly. The script itself contains the complete list of required modules.

container/		
	aescrypt_check	This file will verify the SHA-256 hash value of the AES Crypt source code downloaded during the container build process. This is a security measure to ensure the file was not modified.
	crontab	This is the crontab to install that will control when various Pug processes are executed.
	pug_check	This is an alternative version of the pug_check script to install for containers. This one does not try to check that NTP has a current time, since the NTP service is likely unreachable. However, it is nonetheless very important that the system time is accurate, else files may be prematurely expunged.
	pug_env	This is an alternative version of the pug_env script for containers. It will set certain variables that do not need to be passed in when the container is started.
	pug_exec	Script used to simplify crontab entries and so that stdout and stderr are directed to cron's stdout and stderr so that logging be captured outside the docker (e.g., via docker logs).
	README.md	A file detailing notes about container usage.
	user_pug_env	A template file users may use to populate the pug_env data when loading environment variables via a mounted volume.
docs/		
	Pug System Overview.pdf	This is the document you are reading.
lib/		
	archiveutils.pl	This is a library of functions used primarily by the archiver.
	clientutils.pl	This is a library of functions primarily used by the client utilities in the "bin/" directory.
	cloudutils.pl	This is a library that contains the functions to interface with the cloud storage service provider (presently, only S3 is supported).
	config.pl	This contains a set of global configuration variables that are used by all Pug software components. This script must be modified to contain values that are appropriate for the environment.
	database.pl	This is a library that contains the functions to connect and disconnect from the database.
	fileutils.pl	This is a library that contains functions for use primary with the "files" table and is primarily used by the discover process.
	locations.pl	This is a library that contains functions related to the "locations" table.
	logging.pl	This library contains functions related to logging to syslog.
	utils.pl	This library contains miscellaneous functions, including lock files and string conversions.
schema/		
	pug.sql	This is the database schema file to create an empty "pug" database.

Now having said so many positive things about Pug, it should be appreciated that Pug is not suitable for all tasks. As a tool for ensuring that files are archived and that the data can be safely recovered, Pug

works well. However, Pug is not suitable for backing locations that include empty directories that need to be preserved, backing up special files (e.g., symbolic links), or preserving special permissions (e.g., hidden files on a CIFS-mounted file system). Pug will, however, backup every individual, regular file in a directory (and its subdirectories). Pug will take note of directory structures, but only directory structures for which it maintains archived files.

4 Installing Pug

4.1 Extract the Pug Software

The first step is to select a suitable location for the Pug software. Given that Pug is intended primarily for use by system administrators and should not be run by any user other than the one authorized to access files related to Pug and the data Pug will archive (generally "root"), it is not advised to install Pug in the usual places like /usr/bin/. Rather, it is recommended to select different location, such as /usr/local/pug and ensure that directory has very restricted permissions (e.g., 0700). In that location, put the files in the distribution. For example:

```
$ mkdir /usr/local/pug
$ tar -xzvf pug-1.4.tgz --strip-components=1 -C /usr/local/pug/
$ chmod 700 /usr/local/pug
```

The extraction of the .tgz file will create a subdirectory named using the version of the software release. You can use that subdirectory move the contents to /usr/local/pug/ (or whatever directory you choose for Pug).

4.2 Install MySQL and Create the Pug Database

The next step is to create the MySQL database. If MySQL is already in use, this should be a very simple task. If you do not already have MySQL installed and running, you need to do that, first. Most Linux distributions offer MySQL as a package, so the software is readily available. Installing MySQL the first time will require some effort, so take notes and save your configuration files. Installing it a second time should be much easier. In any case, we will not go into detail on MySQL installation, maintenance, tuning, etc. There are numerous resources on the Internet that do that already. Documentation abounds here: https://dev.mysql.com/doc/. Once you have the MySQL server running, you can create the Pug database with a simple command like this:

```
$ mysql < schema/pug.sql
```

If you do not already have a "user" account defined that Pug can use, be sure to create a database "user" for Pug and give that user full access to the Pug database.

Amazon also has a service called "Amazon RDS" that one could utilize to provide MySQL database services, rather than setting up and managing a database locally. Pug has not been tested with that configuration, but it might be worth exploring if you would prefer to not manage the database installation yourself. You could also use the MariaDB container documented in section 6.2, which is very simple start using.

4.3 Install AES Crypt

Download a copy of AES Crypt™ from https://www.aescrypt.com. You will need a C compiler like gcc or clang installed to make the AES Crypt binaries. With aescrypt-3.14.tgz downloaded, here's what you do to get it installed:

```
$ tar -xzvf aescrypt-3.14.tgz
$ cd aescrypt-3.14
$ make
$ make install
```

Only do "make install" if "make" appears to have worked properly. This will install "aescrypt" and "aescrypt_keygen" in /usr/bin so that it is available to all users and the Pug software. To use AES Crypt with Pug, you will need an encryption key file. To create one, do this:

```
$ aescrypt_keygen -g 64 aescrypt.key
```

The number "64" indicates a "password length" and was selected as this generates a password with more than 256-bits of strength. If you wish to have a longer password, you may specify a password length up to 1024 characters long, but it is really overkill. See this page for more information: https://www.packetizer.com/security/pwgen/.

Now, copy the key to a secure location (e.g., /usr/local/pug/keys) and ensure that only privileged users have access to this key. You may use this same key for the pug_db_archiver (see discussion later) or you may generate a separate key specifically for database archives.

Make sure you do not lose keys. If you lose a key, you will not be able to recover your data! Keys are never stored in the cloud by Pug for security reasons!

Keys should be stored in a secure location accessible only to authorized people. If someone gains access to the keys, they can decrypt your data (assuming they also have access to your cloud storage). Keep backup copies of your keys stored in safe place, preferably encrypted with a password only you or trusted users know. For security from massive disasters, store keys in a place separate from where your data is stored.

4.4 Put the Key File Information in the "Encryption" Table

Now that you have an encryption key you can use for encrypting files, insert a row into the "encryption" table. There is no tool to do this, so you either have to insert it manually using SQL statements or via a management tool like phpMyAdmin (https://www.phpmyadmin.net/) or MySQL Workbench (https://www.mysql.com/products/workbench/).

The contents of **encryption** table are described in section 8.3. Do make sure you put the current time into the row when creating it. You can use the "ptime" utility that came with Pug to give you the current time value to use.

In general, it's good to create new keys periodically. When you create a new key, just generate a new key, put the file in place, and insert a new row. Pug will immediately start using the new key when

archiving any new files if that row has a newer timestamp. ***Old keys cannot be discarded if there is a file stored in cloud storage that uses it.*** It is best to never discard a key.

When putting in the **keyfile** field of the **encryption** table, ensure you use the fully qualified pathname to the key file.

4.5 Define "Locations"

Next, you need to define the "locations" you wish to have Pug archive. In Pug, a "location" is basically a directory on the system. Examples of "locations" might be /home, /home/bob, and /export/nfs/paris. Pug refers to everything by "location" to help make migration simpler. For example, if one of the "locations" to back up is /export/nfs/paris, where "paris" refers to a machine name, and that remote machine's name is later changed to "london", it would be rather frustrating to have to go into the Pug database manually and change all references of "paris" to "london". With Pug "locations", making such a change involves 1) stopping Pug (including the archiver and discover processes) or disabling scanning and archiving for the location (via the "locations" table), 2) unmount "paris", 3) mount "london" (assumption is this is the same set of files, of course), 4) update the "pathname" field in the **locations** table to the appropriate value, and 5) re-start the Pug software (or re-enabling scanning/archiving on the location).

To define a "location", insert a row into the **locations** table. The names and meaning of each field in the table is fully documented in 8.4.

Suppose you want to archive the location /home, you want Pug to scan the directory every four hours to look for modified files, you want do not want to archive any files that start with ~ (often used by Microsoft® Office products to indicate a temporary file), you do not want to archive any directories named "tmp" or "temp", you want Pug to archive any file found 8 hours after it is identified as new of modified, you want to keep the file archived after the user deletes it for 90 days, and you want to allow a maximum of 14 versions to be archived. Let's also suppose you want to archive a shared directory where users put documents. You do not want to archive any .jpg pictures found there. You want to scan it hourly, since it will likely have more changes than /home. But, since people will work on it all day, you would prefer to not archive it until after 12 hours have passed. Also, you want to keep those documents on file for at least one year, but again with a maximum number of versions set at 14. Here's how you configure a "location" to do just that:

path	scanfreq	fileexcl	pathexcl	archivedelay	expungedelay	maxversions
/home	14400	^~.*	/tmp/,/temp/	28800	7776000	14
/export/shared	3600	^~.*,\.jpg$	/tmp/,/temp/	43200	31536000	14

What do those entries mean, exactly? Consider the /export/shared location. Scanning the location every hour means that as each hour passes, Pug notes any modified files. Let's say people in your office work from 8AM to 5PM. Let's also assume the first change was made at 8:30AM and Pug detects that change at 9AM when it happens to make an hourly scan. That file will not be considered for archival until 9PM. During the 9 hours of the day, perhaps many files are changed. Starting at 9PM, the archiver detects that file and any other files that were detected as changed at the 9AM discover run. The archiver

archives those files and stops. It does this through the night until it has archived the last new or modified file.

Should you prefer the "do it all at once" approach, you can also configure Pug to work that way, too. For example, rather than running the scan each hour, you could run the **scanfreq** fields to be "1" (every second) and the **archivedelay** to be "1" (every second), but then only run the discover and archiver processes once, perhaps having discover make a pass over all files at 9PM and then having archiver start at 10PM. If you take that approach, do consider that a failure of some sort might mean nothing gets archived for the day. For example, if the discover process cannot run because the /export/shared filesystem is offline at 9PM, then the archiver will find nothing to archive. If discover happened to work, but your cloud storage provider is having an outage for an hour or so, archiver might fail. So, at the very least, it would be advisable to let scans happen every hour during the night and let the archiver attempt to archive periodically, too.

One thing to note is that you should not have nested "locations". For example, if you define the "location" /home, that will collect all files in the /home directory and all subdirectories. As such, you should not also define a separate location named /home/bob. Nothing will break if you do this, but it just wastes CPU cycles, requires the discover process to scan the /home/bob directory a second time, and inflates the size of the Pug database unnecessarily. It will not result in using more cloud storage space, though, since Pug will recognize that the files found match already-archived files.

An important point to note is that Pug assumes use of UTF-8 in all filenames. It stores all strings in the database in UTF-8 format, too. This is generally a non-issue for those who only use ASCII filenames. However, if you or your users use non-ASCII filenames (including Western European characters that are not ASCII), then it is important to test that those are read by Pug as UTF-8 strings.

If you type "locale" at the Linux command prompt and see some language tag followed by UTF-8, you're probably not going to have any issues if you see non-ASCII filenames properly with the Linux "ls" command. Even so, it's worth doing a small test with Pug to ensure that Pug has the same Locale settings. Those of particular importance are the environment variables LANG and LC_ALL. Those should be set to "en_US.UTF-8", for example. These variables are two-part, with the first part indicating the language (which is less important) and the second part indicating the "code page" (UTF-8). It is the UTF-8 "code page" setting that is important.

An easy way to test the behavior of Pug is to create a file that uses non-ASCII characters and ensure that it looks correct in the "files" database and that the **files.pathhash** value is correct for that file. You can use this command on many Linux systems to see the SHA-1 hash of the pathname:

```
echo -n "filename" | sha1sum
```

For "filename" above, use the value you see in the **files.pathname** field in the Pug database. The response you get back should match the **files.pathhash** value. (Note the -n flag on echo means "do not add a trailing newline". The proper flag used with echo varies by Unix variant, but Linux seems to have settled on this syntax.)

Why is Pug designed to assume only UTF-8? Quite frankly, it is a pain to deal with numerous different character encodings and UTF-8 appears to be the clear winner in terms of widely used character encodings for filenames and the Internet in general.

4.6 Modify the pug_env File

Modify the lib/pug_env file so that it contains the proper values. This file controls the behavior of the Pug software (e.g., what database to connect to and where to store files in the cloud). It is consumed before invoking any other Pug software routines. Given the sensitive nature of some of the parameters contained in this file, it should be installed with restricted access.

Variable	Description
PUG_SCRIPTS	This value will be prepended to the PATH environment variable so the system can find the Pug software modules. It should include both the bin/ and sbin/ directories.
PUG_DATABASE_USER	This is the MySQL database "user" identifier that Pug should use.
PUG_DATABASE_PASSWORD	This is the MySQL database password associated with the above user ID.
PUG_DATABASE_NAME	This is the name of the MySQL database to access.
PUG_DATABASE_SERVER	This is the hostname of the MySQL database. By default, this value is set to "localhost", since it is assumed the MySQL database will be local. However, the MySQL database could be installed on an entirely separate machine.
PUG_DATABASE_SSL	If set to 1, then SSL will be used for database connections. A value of 0 indicates that SSL is not used. (SSL is an historical name. Modern systems use TLS for connections, but changing the parameter name is confusing since underlying software APIs still use the term SSL.)
PUG_AWS_ACCESS_KEY_ID	This is the Amazon S3 "Key ID" value to use.
PUG_AWS_SECRET_KEY	This is the Amazon S3 "secret access key" to use.
PUG_AWS_S3_BUCKET	This is the name of the Amazon S3 bucket to use. This bucket must already exist. Pug will not create or delete buckets.
PUG_AWS_S3_FILE_PREFIX	When archiving files to cloud storage, the storage location might be used by other software, including other Pug instances. This prefix, which MUST be assigned, helps to differentiate Pug files from other files or one server running Pug from another. This may be set to "pug" (default) or the name of a server like "paris".
PUG_TEMP_STORAGE	This is a directory where Pug can create temporary files. This directory should be local to the machine (as files are copied to this location, compressed, and encrypted), have sufficient storage space to store at least a few copies of the largest file that will ever be archived, and be secured such that nobody can access this except authorized users. Every file that is archived in cloud storage passes through this directory. Thus, it's advisable that this directory be owned by "root" with

	permissions set to 0700. Do not use a directory that might be used for any other purpose.
PUG_DATABASE_ARCHIVE_KEY	When the database is archived via the pug_db_archiver process, this is the AES Crypt encryption key that will be used to encrypt the file before uploading to cloud storage. Note that the path to the encryption key used for all other archive files is stored in the database. While the archiver could have used a key referenced in the database, having it explicitly listed here helps avoid any confusion about which key was last used to archive the database in the event of disaster recovery.

4.7 Explanation of the config.pl File

The lib/config.pl file contains a lot of system-wide configuration parameters, many of which are taken from the environment variables listed in the previous section. Additional parameters *may* be modified directly in this module, though care should be exercised when doing so. Every configuration parameter is defined below. Note that all of the variables start with "$main::". This is Perl syntax to indicate that the variable is global. Do not change that. Also, if strings are not inside quotes, there may be a reason. Do not add or remove quote marks around parameter values.

Parameter	Description
database_user_id	The environment variable PUG_DATABASE_USER.
database_password	The environment variable PUG_DATABASE_PASSWORD.
database_name	The environment variable PUG_DATABASE_NAME.
database_server	The environment variable PUG_DATABASE_SERVER.
database_use_ssl	The environment variable PUG_DATABASE_SSL.
aws_access_key_id	The environment variable PUG_AWS_ACCESS_KEY_ID.
aws_secret_access_key	The environment variable PUG_AWS_SECRET_KEY.
aws_s3_bucket_name	The environment variable PUG_AWS_S3_BUCKET.
pug_temp_storage	The environment variable PUG_TEMP_STORAGE.
pug_archive_empty_files	If this value is "Y" then Pug will archive files, even if they have a file size of zero octets. The default is to archive zero-length, but you may set this to "N".
pug_log_to_syslog	When programs in the sbin directory run, they log messages to syslog by default. If you would also like to have the output displayed on the screen (useful for when these commands are run manually or when you wish to see output from cron jobs), set this value to "N". The default is "Y".
pug_archive_file_prefix	The environment variable PUG_AWS_S3_FILE_PREFIX.
pug_archive_file_prefix_separator	Files stored in the cloud are named pug/10353.1, for example. The "pug" at the front is the aforementioned prefix. There is a slash (the default "separator") followed by numbers that refer to the archive record and file part number. In Amazon S3, this form simulates "folders". This is purely a matter of preference, but for large sites that use Pug to back up many servers, using the "/" style separator of even a separator like

	"/<server_name/" might be useful. Using other characters are separators (e.g., "-") is perfectly valid.
pug_upload_part_size	Uploading to cloud storage can sometimes be challenging when uploading large files. Therefore, Pug breaks files into "parts" that are this size and uploads each "part" separately. That ".1" in the example above is the "part" number. By default, this value is set to 20MB. If Pug uploads a file that is 100MB, it will be done so in 5 parts. This value may be changed at any time, though be mindful of the fact that Pug reads this amount data from large files and might maintain multiple copies of that data in RAM during archival operations. Uploading files using a large "part" size also provide little or no performance gains. For performance reasons, values lower than 8MB should be avoided.
pug_archive_max_sequential_bytes	If this value is greater than zero, it indicates that maximum number of octets to the archiver will store in the cloud before exiting. This value is checked after successfully uploading each complete file, as the archiver will never upload a partial file.
pug_archive_expunge_delay	This parameter tells Pug how long it should wait before expunging an archive file from cloud storage after it detects that the file has been deleted locally and there are no files (including old "deleted" files) referencing it. The default value is one day (86400 seconds), though having a file sit in cloud storage for a week or two is not a bad idea. That said, a longer time period is not necessary since expunging is better controlled via the **xpungedelay** parameter defined for each location in the locations table.
pug_delete_xstatus_delay	As database records related to files, cloud archive files, and archive parts are "deleted", they are not really deleted. Rather, they are marked with an "X". This is useful in case the administrator wants to look for information related to a specific file or look at the file's history. However, there is a point where the value diminishes. This parameter indicates how many seconds should pass before the housekeeper deletes those records permanently from the database. The default is 31536000 seconds (about one year).
pug_default_directory_create_mask	When Pug needs to re-create a directory as part of the file restoration process, it will first create a directory using this mask and the directory will be owned by the process running the "pget" or "precover" command (e.g., "root"). This value should be restrictive so that information is not leaked. The default value is 0700, meaning only the user running pget or precover can access the directory. Proper directory ownership and permissions will be restored if Pug can locate that directory in the Pug database.
pug_db_archive_key	The environment variable PUG_DATABASE_ARCHIVE_KEY.

pug_log_app_name	All pug_* processes log to syslog and using the name of the application. This variable exists here only as a fallback should some service not have this assigned. Actual syslog entries would contain things like “pug_archiver”, not just “pug”.
pug_log_facility	This is the syslog “facility” to use. By default, it is LOG_USER.

5 Using Pug

Once installed and configured, you can start using Pug right away. Since there are several configuration parameters and it is easy to make a mistake the first time, it is recommended that you start using Pug on a small scale (e.g., just one location with a few small/medium size files). Run the discover and archiver programs manually to see how they work. Delete some local files (do make sure these are not important files!) and test that you can get them back from cloud storage using “pget” or “precover”. Use the “pls” command to see the files presently archived in cloud storage.

Modify a file and see that a new version gets archived. The “pls” command should show you two versions of the file.

5.1 Running Pug from cron

Once you feel relatively content that Pug is working for you, consider running these processes via cron by inserting these lines into the crontab file for root:

```
#MIN    HOUR   DOM MON DOW COMMAND
*/15 0-4,7-23   *   *   *   . /path/to/sbin/pug_env; pug_discover
*/15 0-4,21-23  *   *   *   . /path/to/sbin/pug_env; pug_archiver
15       5      *   *   *   . /path/to/pug_env; pug_housekeeper; pug_db_archiver
45       6      *   *   0   . /path/to/pug_env; pharmony
```

The first command on each line will source the pug_env file to ensure the PATH and PERL5LIB values are set properly, then it will execute pug_discover, pug_archiver, pug_housekeeper, or pub_db_archiver. Running the database archiver is recommended after the housekeeper completed. There is also a process scheduled once per week called pharmony which will double-check that every file that should be in cloud storage is and that there are no unreferenced files in cloud storage. While this should never be an issue, this serves to double-check that the database and what is stored in the cloud are in harmony.

The reason for running these commands frequently is so that files are uploaded somewhat evenly during the day, or ideally during the night. (This cron job schedules all uploads to start at 9PM and continue until just before 6AM). When a new or modified file is seen by discover, it inserts a record into the database. However, pug_archiver does not upload that immediately. It is required to wait at least **locations.archivedelay** seconds before initiating the upload. The next time the archiver runs, and it sees a file that satisfies this time, it will upload it. If you have a place of business where users create files from 8AM to 6PM, for example, you probably want to insert a 10-hour **archivedelay** (36000), so that the first file seen at 8AM will not get archived until at least 6PM. Each hour during the night, the archiver selects

the next set of files that are eligible for archival. If everything works, all files will be archived before the next business day and uploaded at a steady pace.

Note that directories are not necessarily scanned every 15 minutes. You can control the rate at which a directory is scanned using the **locations.scanfreq** value. So, while discover might run every 15 minutes, if there are no directories that are scheduled to be scanned, it will exit immediately.

You can adjust the time cron runs these commands, the **locations.scanfreq**, and **locations.archivedelay** to best suit your needs.

5.2 Massive Initial Archival

Please be advised that if you have tens of thousands or hundreds of thousands of files to archive, the first run of the archiver might take an incredibly long time to complete. By design, no two instances of the archiver will run at the same time: it is just a serial process of uploading files to cloud storage.

What you might prefer to do if you have a very large number of files is to "archive" them to local storage. This would require modifying the cloudutils.pl library to store objects to some local storage device, such as a USB-attached external hard drive. Once all files are archived using Pug, the drive can be shipped to Amazon and contents installed in S3. Of course, this would require changes to the Pug code and a good understanding of what Amazon will do with the data, but this is a possible solution to the massive initial archival issue.

5.3 Multiple Instances of Pug

Running multiple, separate instances of Pug are certainly possible. To do that, install the software in some location (e.g., /usr/local/pug), but place the pug_env file in a different directory. Since the name of the database and all other instance-specific data is isolated to that pug_env file, this is the only file that serves to differentiate unique characteristics of one Pug instance from another. One might use one Pug instance to archive normal user data to a normal S3 bucket and another to archive lesser important data to a different S3 bucket, perhaps having different service guarantees.

Note that when the discover or archiver process run, they create a lock file in the temporary file directory specified by the environment variable **PUG_TEMP_STORAGE**. This could cause problems for multiple instances of Pug if they share the same directory, so specify a different temporary directory for each instance. For example, /usr/local/pug/tmp/paris and /usr/local/pug/tmp/london as two different temporary file directories.

6 Using Pug via a Container

Using a container may simplify running Pug, especially when using a NAS that supports containers. This section will demonstrate how one might create such a deployment where there is a single database container, one web management container running phpMyAdmin, and two Pug containers that archive two different sets of data that gets stored in two different Amazon S3 buckets.

Examples in this section utilize Docker, though alternative tools (e.g., Podman) may be used.

If you have no interest in containers, you may skip this section entirely.

6.1 Creating a Container Network

The Pug-related containers will run within the confines of a user-defined bridge network that, for the purposes of illustration, we will call the network “pug”. To do that via the command-line, type:

```
$ docker network create pug
```

6.2 Creating a Database

The second step in the process is to create a database server. For this example, we will use the container offered by the MariaDB Foundation. They publish their containers via Docker Hub here: https://hub.docker.com/_/mariadb/.

The first thing to consider is where data will be stored. You may use a local directory dedicated to holding the MariaDB data or you might wish to use a container volume. For this example, we will do the latter. First, create a volume called “pug_db”:

```
$ docker volume create pug_db
```

Once the volume is created, launch a new container running MariaDB. In this example, we will use MariaDB 10.6.4, name the container pug_db, create a user named “pug”, and create an initial database also called “pug”. The passwords for the root and pug accounts are both specified via the command-line, though you might want to consider something more secure.

```
$ docker run --restart=always --network=pug --name=pug_db -e
MARIADB_ROOT_PASSWORD=secretsauce -e MARIADB_DATABASE=pug -e MARIADB_USER=pug -e
MARIADB_PASSWORD=secretpooch -vpug_db:/var/lib/mysql -d mariadb:10.6.4
```

You can now execute commands to see that the pug database did, indeed, get created:

```
$ echo 'show databases;' | docker exec -i pug_db sh -c 'exec mysql -uroot
-psecretsauce'
```

To see that the user “root” and user “pug” were created, execute the following:

```
$ echo 'SELECT User, Host, Password FROM mysql.user;' | docker exec -i pug_db sh -
c 'exec mysql -uroot -psecretsauce'
```

You can also verify that the user “pug” exists and has privileges for the table “pug”:

```
$ echo 'show grants for pug;' | docker exec -i pug_db sh -c 'exec mysql -uroot
-psecretsauce'
```

Should you wish to create another database on this server for a different instance of the Pug software (e.g., to backup data to a different S3 bucket), we can create a different database on this same Docker container. Let’s say you have data that you wish to archive in a lower-cost S3 storage bucket. For simplicity, we will call this new database “archive”. To do that, you can issue the following command:

```
$ echo "create database archive;" | docker exec -i pug_db sh -c 'exec mysql -uroot
-psecretsauce'
```

Once the database is created, the user “pug” needs to be granted access. If you wish, you can also create a different user, but we’ll use the previously created user “pug” for simplicity:

```
$ echo 'grant all on `archive`.* to `pug`@`%`;' | docker exec -i pug_db sh -c
'exec mysql -uroot -psecretsauce'
```

And, once again, you can run the same commands above to check that the database was created and that the user “pug” has been granted privileges on both.

Before going further, you might wish to stop the container (e.g., via “docker kill”) and re-start it to verify that the database storage is not destroyed with the container:

```
$ docker kill pug_db

$ docker run --restart=always --network=pug --name=pug_db
-vpug_db:/var/lib/mysql -d mariadb:10.6.4
```

You can run the same “show databases” and “show grants” commands to verify the databases and permissions persisted.

Note that the database schema in the schema directory of the Pug software distribution will create a database named “pug” if it does not already exist. To use that same schema file with a different database (e.g., “archive”), you will need to edit the file and change the “CREATE DATABASE” and “USE” lines near the top of the file from “pug” to “archive” (or whatever you call your other database). While creating the container created a database, the database has no schema. To load the schema for Pug, execute the following for the “pug” database from within the “schema” directory:

```
$ docker exec -i pug_db sh -c 'exec mysql -uroot -psecretsauce' <pug.sql
```

Now the schema should be fully loaded.

```
$ echo "use pug; show tables;" | docker exec -i pug_db sh -c 'exec mysql -uroot
-psecretsauce'
```

You should see the various tables referenced in section 8 listed in the output.

6.3 Populating the Database

You may issue SQL commands to populate the database like the ones issued in the previous section, though using a web interface is much simpler. For that, we will create a phpMyAdmin container like this:

```
$ docker run --restart=always --name pug_myadmin --net pug -p 8080:80 -d -e
PMA_HOST=pug_db phpmyadmin:5.1.1-apache
```

This makes the phpMyAdmin interface available from the machine running docker on port 8080. You can log in either using the previously created “root” or “pug” accounts. The phpMyAdmin software will connect to the database server on the “pug” network (the host we named “pug_db”) using the credentials you enter when prompted.

NOTE: The default phpMyAdmin container does not use TLS for securing connections between your web browser and the container.

Through this web interface, you can easily populate the **locations** table and the **encryption** table. Those are the two tables that must have valid data before attempting to run the Pug container.

6.4 Creating the Pug Container

To run, Pug needs several environment variables defined (enumerated in the pug_env file). It also needs to have AES Crypt keys. Ideally, those encryption keys would be stored on a local volume (outside of the container system's management) so that they can be backed up and new ones easily added. When performing disaster recover, you will need access to those keys.

With respect to the Pug environment variables, some are populated via the container/pug_env file installed by the Dockerfile as the container is created. Those can remain unchanged (and probably should be) since those only affect the behavior of Pug inside the container. For example, the PUG_TEMP_STORAGE variable can safely have a fixed location inside the container, as it will not conflict with and other containers or local storage. However, those variables that are specified to a running instance (e.g., the database name, credentials, etc.) must be set when the container is started. There are two ways to do that, the first is by using the "-e" flag given to Docker for each variable. Alternatively, one may place the variables in a pug_env file that is read by Docker container's pug_env file. This user-provided pug_env file is expected to exist at /pug/bin/pug_env.

When running in a Pug container, the set of variables that must be set include:

Variable name	Comment
PUG_DATABASE_USER	
PUG_DATABASE_PASSWORD	
PUG_DATABASE_NAME	
PUG_DATABASE_SERVER	
PUG_DATABASE_SSL	Defaults to 0 (i.e., no SSL)
PUG_AWS_ACCESS_KEY_ID	
PUG_AWS_SECRET_KEY	
PUG_AWS_S3_BUCKET	
PUG_AWS_S3_FILE_PREFIX	Defaults to "pug"
PUG_DATABASE_ARCHIVE_KEY	

Note that those with default values in this table need not be set. You can use that as a source for your own pug_env file by adding the variables you need to define and putting them in /pug/bin/pug_env or you may pass each as an environment variable using "-e" flags, as discussed above.

As stated in the first paragraph, AES Crypt keys should also be provided via this mounted volume. This is another reason why it might be easier to create a local directory mounted as a volume having a structure like this:

```
/pug/
    bin/
```

```
keys/
dump/
```

This directory would be mounted as a volume via the "-v" option like "-v /path/to/dir:/pug" to /pug in inside the container.

The Pug software is run by cron inside the container. In the container/ directory, you will see a crontab file with the times that Pug should perform various tasks. You may keep these defaults or change them for your environment. Note the system's time inside the container is expressed in UTC time, so the hours shown in that file are in UTC time. You will also notice that the entries use a script called "pug_exec" to simplify invoking Pug and to ensure output is directed appropriately so that "docker logs" can be used to see the output (success or failure) of scripts. This is necessary, as crond (the cron daemon) forks a child process and output to that child are not captured by Docker. This script redirects that output to the parent crond's stdout and stderr file descriptors so logging works are expected.

Note that after the db_archiver runs, db_dump is executed in the crontab. This will dump the contents of the database to /pug/dump/ for local backup.

To build the Pug container, execute this command:

```
$ docker build -t pug .
```

This is a two-stage build process. The first stage builds AES Crypt and verifies the SHA-256 hash of the file that is downloaded. The second stage then copies the AES Crypt executables into the final Pug container image. Note that there may be an intermediary image created that you can safely delete. Such dangling images appear with the name "<none>" when issuing the "docker images" command. As a tip, you can easily remove all such images via this Docker command:

```
$ docker rmi $(docker images -f "dangling=true" -q)
```

Now it is time to run the container. For our example, we will assume the required environment variables are in a directory having a structure outlined above. Let us also assume you want to backup the data in /home and configured the MariaDB database with the "location" /export/home as the path to that data. Given those assumptions, we can start the Pug containers using a command like this:

```
$ docker run --restart=always --name pug --net pug -d -v/path/to/pugdir:/pug
-v/home:/export/home pug:latest
```

You can mount as many "locations" to back as you wish up as volumes using the "-v" flag.

At this point Pug should be running and communicating with the database container. Monitor for errors using a command like this:

```
$ docker logs -t -f pug
```

6.5 Secondary Database Backup

While the Pug container will, by default, upload the current database to the cloud weekly, it also calls pug_db_dump to dump the database to local storage via cron. If you would prefer to not do that, you

can remove that part of the cron job. However, it is recommended to back up the database locally, nonetheless, and preferably to a different host. It only takes a moment for relatively small datasets (even hundreds of thousands of files), so performing this step daily is reasonable. Using the example container and database named “pug” from previous sections, this can be done like this:

```
$ docker exec pug_db mariadb-dump -B pug >/path/to/backup/pug.sql
```

The output may also be directed to gzip, aescrypt, or other tools before being written to disk. If you wish to encrypt the file and compress it, it’s recommended to compress it first since encrypting the file will make it nearly impossible to get good compression due to the random nature of encrypted files.

See the mariadb-dump command for more options or see the pug_db_dump script as an example.

7 System Operation

7.1 Identifying Files to Back Up

The discover process is responsible for identifying potential files to be backed up. It does so by searching each defined “location” for new, modified, or deleted files. It notes any such files in the Pug database and then collects a list of directories and their associated ownership/permissions. The latter is used in restoring proper permissions and ownership should files need to be restored from cloud storage.

Files are inserted into the “files” table with a status of “N”. At some point when the archiver (pug_archiver) executes, it looks for files with a status of “N” and then checks to see that the current time is greater than the value of the file’s “stime” value (value when the “N” status was applied) plus the **locations.archivedelay** value. If so, then the file is archived.

Files noted as deleted will be updated in the “files” table with a status of “D”, which schedules the file for later deletion.

7.2 Marking Files for Future Deletion

When a file is deleted, the discover process takes note of those deletions. When the discover process sees a file that is apparently deleted, it marks the file as “deletion scheduled” (status “D”) in the database. However, this does not actually delete the file from cloud storage. If you issue the “pls” command, you will still see the file. It would seem to be available, and that’s because it is.

After the file has been scheduled for deletion for more than the length of time specified in **locations.expungedelay**, the housekeeper will mark the file as expunged (status “X”). It is only at that point that the file will no longer be visible via “pls”.

Note, though, that if there are two identical files on the system, one is deleted and the other is not, there would be only one copy of the file in cloud storage, and it would still be marked as archived (“A”) in the database table “archive”. The two files would both share the same **akey** value, which refers to the **archive.skey** field. Only after *all* copies of the same file are deleted will a file be scheduled for deletion from cloud storage.

Once it is determined that there are no longer any files referring to a specific archived file, the archived file will be scheduled for deletion by the housekeeper. It will not identify files for deletion and delete them in one pass. The next time housekeeper runs and after the “pug_archive_expunge_delay” (in config.pl) delay time has passed; the housekeeper will remove the file from cloud storage.

In short, Pug does not get in a hurry to delete files. And this is the way it should be. One should always have time to recover a file accidentally deleted. And, if you are of the mind that a file should never be deleted, just set the **locations.expungedelay** value to 0 to prevent any files in the specified location from ever being expunged. Likewise, if you wish to prevent any files stored in cloud storage from ever being deleted, set the “pug_archive_expunge_delay” value to zero. (Note, though, that if you do this, the only way to map a particular file in the cloud storage to a file on the local system will be through the records marked “X” in the “files” table. So, if you set “pug_archive_expunge_delay” to zero, you should also set “pug_delete_xstatus_delay” to 0.)

By default, files marched to be expunged are deleted after 24 hours. The rows in the database having a status of “X” are deleted after 365 days. Generally, these parameters should not be tuned in config.pl and, instead, deletion of files should be controlled by the **expungedelay** value in the **locations** table.

NOTE: If “pug_discover” scans a “location” and finds no files, it will *not* continue processing the location looking for files to mark as deleted. This is for data safety reasons. If a filesystem is not mounted or if all files were inadvertently removed, it would be unfortunate for Pug to interpret that as meaning all files have been deleted and schedule them for deletion from cloud storage. If you wish to have Pug delete all files in a location, update the files table by changing the status of all files in that location from “A” to “D”. Pug will then remove files as it would normally do.

7.3 Recovering Files

If a user deletes a single file or a few files, the easiest way to recover those files is with the “pget” command. The “pget” command has the following syntax:

```
usage: pget { -f <file-id> | -l <location> -p <pathname> } [<new_filename>]
        -f - retrieve a file by File-ID shown in pls
        -l - the "location" from which to retrieve the file
        -p - the "pathname" of the file within that location
```

If you know the File-ID, that’s the simplest way to get the file. The File-ID is displayed next to the files shown with the “pls” command. If the target file has a File-ID of “537”, then you can recover the file to its original location by issuing this command:

```
pget -f 537
```

If you wish to retrieve the file, but place it in a different location, just specify the new pathname as the next item on the command like, like this:

```
pget -f 537 /tmp/somefile.txt
```

If you lose all the files in a “location” or a subset of those files, you will use the “precover” command. The “precover” command has the following syntax:

```
usage: precover -t <time> -l <location> [<path_prefix>]
        -t - recover all files that were present at this time or later
        -l - the "location" for which files need to be recovered
```

The -t parameter basically means to “recover all files that were present on the system on or after this time”. Do not set this value to zero or some very old time, otherwise the precover command will restore a bunch of files that the user deleted ages ago. The only thing more dissatisfying to a user than losing files is having a bunch of old files reappear that they had long since deleted.

Let’s say you have a location named /export/nfs/paris and the server’s hard drive crashed two days ago. The server is now back online and you want to restore all of the files. It’s reasonable to suggest that we should recover all files that were present on the system as of three days ago. Perhaps we get a few extra “deleted” files, but that is not unusual in any file recovery scenario. But what is the Unix system time “three days ago”? Pug has a utility for that called “ptime”. Just type this:

```
ptime -d 3
```

The program will tell you the current time and what the time was three days ago. Let’s say the time three days ago happened to be “1357118980”. To recover all files in location /export/nfs/paris from three days ago, enter this command:

```
precover -t 1357118980 -l /export/nfs/paris
```

This will cause all archived data files to be pulled out of cloud storage and placed back into their original location. Pug will try to restore owner, group, permissions, and mtime on each file and owner/group and permissions on each directory.

If only a subset of files, say those in the directory “/export/nfs/paris/documents/contracts/louvre/”, then all those files could be recovered via this command:

```
precover -t 1357118980 -l /export/nfs/paris documents/contracts/louvre/
```

Note that the “path prefix” part of the command excludes path to the “location”.

7.4 Recovering from a System Failure

To recover from a total system failure, one must re-install the Pug database and restore the data.

In the event of a total system disaster, one must restore the database manually. If you have a dump of the database on a secondary machine, then you can use that file to restore the database.

If you use the “pug_db_archiver”, the archive copy needs to be restored from cloud storage. The location of the archive in cloud storage is part of the configuration parameters PUG_AWS_S3_BUCKET and PUG_AWS_S3_FILE_PREFIX. Let’s assume the bucket name is “mypug” and the archive file prefix is “machine1”. Then the database file would be stored in Amazon S3 with the following “object” name

"mypug/machine1/ db.sql.gz.aes.*n*" where *n* is a number indicating the archive part number. Small archives will have a single part, so you will only see the value 1. Larger archives will be broken into several parts. Use a tool like s3cmd or other to view the files in Amazon S3 to see database archive files and pay close attention to the file timestamps. (If you see an archive part with an older timestamp than the .1 part, then you should ignore those subsequent part numbers as the part is of no value and is left over from a time when the database was larger in size than the most current backup of the database.) To use s3cmd to see the list of files, type this (using the example values):

```
$ s3cmd ls s3://mypug/paris/db.sql.gz.aes
```

This will show you every part number. Download each part using a command like this:

```
$ s3cmd get s3://mypug/machine1/db.sql.gz.aes.1
```

As explained above, if you see a higher-numbered part number that has a timestamp older than .1, ignore that part and all subsequent part numbers. You may even safely delete those additional part numbers, but do not do it until you know you have successfully restored the database.

Now, assemble the part numbers in order, like this:

```
$ cat db.sql.gz.aes.1 db.sql.gz.aes.2 > db.sql.gz.aes
```

Now decrypt the file using your archive encryption key (defined by the **PUG_DATABASE_ARCHIVE_KEY** environment variable). Let's assume the key is called "mysecret.key". Use "aescrypt" to decrypt and "gunzip" to decompress the file as follows:

```
$ aescrypt -d -k mysecret.key db.sql.gz.aes
$ gunzip db.sql.gz
```

You can use the db.sql file to restore the database using the "mysql" command that is part of the MySQL distribution.

Once the database is restored, the Pug software is re-configured and ready to run, you can restore each of the lost locations as described in the previous section.

As a final comment on file recovery, one should not try recovering large numbers of files while at the same time Pug is trying to archive files. During any large file recovery operation, it is best to disable Pug. You have several ways to do that:

- Comment out the lines in crontab
- Modify the sbin/pug_check script to always return the value 1
- Modify the locations table of affected locations and set the **scanfreq** and **archivedelay** values to zero

Whatever approach you take, be sure to undo that once the data is recovered so that Pug can again continue archiving files.

8 Database Schema

There are several database tables defined to support the Pug software. They are each presented in this section with an explanation of each table and column. Every text field is defined to be a UTF-8 string. All unsigned integers (uint) are 64 bits in length.

8.1 "Archive" Table

The "archive" table is used to store a list of all files that are archived in the cloud. Any two files that have the same hash value are presumed to be identical files and are uploaded only once. Once a new file is added to the "archive" table and uploaded, newly discovered files that match these archived files are merely linked via the relationship **files.akey = archive.skey** and not uploaded again.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each unique file uploaded to the cloud storage
hash	char(40)	The SHA-1 hash of the original file
size	uint	The size of the original file in octets
ekey	uint	Encryption tool and key used to encrypt file (references the "encryption" table)
uhash	char(40)	This SHA-1 hash of the uploaded file, which will be different than the original file due to compression and encryption
usize	uint	Size of the uploaded file, which will be different than the original file due to compression and encryption
status	char(1)	File status, which may be one of: A) Archived D) Deletion scheduled, but the file remains in the archive until the housekeeper runs and sees that the archive expunge delay time has passed R) Archive scheduled for removal and transitions to expunged status once all associated archive parts are expunged. U) Uploading, not ready for retrieval X) Expunged
stime	uint	Time when the status was last changed (or refreshed in the case where large files are uploaded in parts)
timestamp	uint	Timestamp when this entry was created

8.2 "ArchivePart" Table

If files are of significant size, then uploading an entire file to some cloud storage services in a single unit can fail. Amazon's S3 service offers a means of breaking object uploads into pieces, but not all services offer this feature and so Pug does not rely on that feature. Pug will handle uploading files in smaller parts that are stored as individual objects in cloud storage. Files larger than the configured "pug_upload_part_size" size will be split into parts. The default value is 20MB, though any reasonable value may be specified. (Note that this value may be changed at any time without affecting previously stored files). When uploading files, the name given to an object part looks like <prefix>-<archive.skey>.<archivepart.part> (e.g., "pug-35565.1). Note that the entire part is loaded into RAM

when uploading or downloading, so the size of “pug_upload_part_size” should not be excessive. Also, the hyphen separator character may be specified in the config file to be something other than a hyphen.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each unique file part uploaded to the cloud storage
akey	uint	The skey of the row in the archive table to which this part belongs
part	uint	The part number (from 1..n) of the uploaded file
hash	char(40)	The SHA-1 hash of this part
partsize	uint	The size of this part
status	char(1)	File status, which may be one of: A) Archived U) Uploading, not ready for retrieval X) Expunged
stime	uint	Time when the status was last changed
timestamp	uint	Timestamp when this entry was created

8.3 “Encryption” Table

This table specifies the encryption tools used and associated keys. Support for a specific tool is built into Pug, with support presently only for AES Crypt. When archiving files, Pug will select the row with the most recent timestamp value. To get the current time from Linux, use the include “ptime” utility.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each combination of encryption tools and encryption keys
tool	char(1)	Encryption tool used to encrypt, which is one of: A) AES Crypt with a 256-bit key
keyfile	varchar(256)	The pathname to the key file used for encrypting stored files
timestamp	uint	Timestamp when this row was created

8.4 “Locations” Table

This table specifies the locations (directory paths) that should be searched for files to be archived in cloud storage. Pug will periodically make a pass over each location and take note of any new or deleted files. Additions and deletions are noted in the “files” table.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each location
path	varchar(256)	The pathname for the location to archive (e.g., “/home”)
scanfreq	uint	Number of seconds that must elapse since lastpass before this location will be searched again (e.g., 3600 for one hour). If this value is zero, the discover process will skip this location when looking for files.
lastpass	uint	The time this location was last searched
fileexcl	varchar(256)	This is a comma-separated (“,”) list of regular expressions used to match file names to be excluded from archival. If a comma happens to be in the name of the file to be excluded, it may be

		preceded by \ to protect it. Since \ is used to protect it, then \\ is required to represent a single \ character. Valid examples include "^Thumbs.db$" (a specific filename), "\.jpg$" (all files ending with .jpg), "^~.*" (all files that start with a ~ character). If all of these were used together, this field would have the value "^Thumbs.db$,\.jpg$,^~.*". These expressions are case-sensitive, as are filenames on Linux systems. Note that it is illegal to use "*.jpg" since "*" is used to match the previous character in a regular expression; syntax as shown previously must be used instead. This column only applies to the file collection process (i.e., determining what files to archive). Pug will produce an error if patterns are invalid. These strings are case-sensitive.
pathexcl	varchar(256)	This is a comma-separated (",") list of pathnames to exclude from archival. If a comma happens to be in the name of the file to be excluded, it may be preceded by \ to protect it. Since \ is used to protect it, then \\ is required to represent a single \ character. The pattern match is performed against the relative pathname (including the filename) of the target file relative to the "path" field in this row. If you wish to exclude any pathname that contains "@eaDir" as a subdirectory, then "/@eaDir/" would do that. If you wish to exclude all files ending in .jpg, then "\.jpg$" would do that. To exclude both, this field would contain "/@eaDir/,\.jpg$". Note that it would be illegal to use "*.jpg" since "*" is used to match the previous character in a regular expression; syntax as shown previously must be used instead. This column only applies to the file collection process (i.e., determining what files to archive). Pug will produce an error if patterns are invalid. These strings are case-sensitive.
archivedelay	uint	The delay in seconds from the time a file in this location is observed (inserted with an 'N' status) and the time the file will be considered for archival. Having a value equal to a half day or longer will help avoid archiving files that are created temporarily or renamed. If this value is zero, the archiver process will skip this location when archiving files.
expungedelay	uint	The delay in seconds that should be imposed after a file has been deleted locally before it is expunged from cloud storage (e.g., 86400 for a day, 2592000 for 30 days, 7776000 for 90 days). The

		purpose for this field is to allow recovery of a deleted file for a period of time from the archive. It should be large enough to ensure no file is deleted permanently before recovery can be attempted. If this value is zero, the housekeeper will never expunge files scheduled for deletion files. Rather, they will remain in the "D" state indefinitely.
maxversions	uint	Some files change daily or weekly and, while we may want to retain a deleted file for a certain period of time, we may wish not to retain too many copies. This field indicates the maximum number of "deleted" versions of a file to retain within the **expungedelay** timeframe. If this field is set to 5, for example, then if a sixth file appears with the same name, the oldest copy will be expunged next time the housekeeper operates. If this value is zero, Pug will not enforce a maximum number of versions, thus only the **expungedelay** will be considered when expunging files.
timestamp	uint	Timestamp when this row was created

8.5 "Files" Table

The "files" table is used to record all files that are found and the status of those files. Details about each file, including user/group ownership, permissions, and modification date are all recorded in this table.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each file
akey	uint	A reference to the skey field in the archive table for this file, or zero if the file has not been archived
location	uint	A reference to the location in which this file is stored
pathname	varchar(256)	This is the relative pathname of the file, relative to the "location" in which the file is stored
pathhash	char(40)	This is an SHA-1 hash of the pathname, which is used to identify all files having the same name
owner	varchar(256)	The name of the owner (converted from the numeric UID)
group	varchar(256)	The name of the group (converted from the numeric GUD)
mode	char(4)	This is a textual representation of the file permissions. It is an octal number where the first digit indicates the SUID(4)/SGID(2)/sticky(1) bits, the second octet is the owner permission for read(4), write(2), execute(1), the third octet is the group permissions, and the last octet is the "other" permissions. For example, "0644" means that the file has owner read/write, group read, and other read permissions.
size	uint	The size of the file in octets
mtime	uint	The file modification time, this along with the file size helps Pug decide when a file on the system may have changed
status	char(1)	File status, which is one of: N) New or modified file that needs to be processed by the archiver. If an older version of this file sharing the same

		pathname exist with in an active ('A') state, the archiver will change the status to 'D' upon archiving this new version. A) Archived D) Deletion is scheduled, but the file is still available in the archive (files remain in this state until expunged or recovered) R) File has been scheduled for removal, but is not yet expunged. Files scheduled for removal might move to expunged state immediately or might be removed later by the housekeeper. X) File expunged (or otherwise forever gone). If Pug expunges a file permanently from the cloud storage archive, the file status is changed to 'X'. Likewise, if Pug identifies a new file ('N'), but it is apparently deleted before the archiver can archive it, the status moves directly from 'N' to 'X'.
stime	uint	Time when the status was last changed, used primarily in deciding when to expunge files (just useful information otherwise)
timestamp	uint	Timestamp when this row was created

8.6 "Directories" Table

The "directories" table is used to record all directories that are along the path of any archived file. This table records the last known owner, group, and permission information so that directories can be re-created with the correct permissions along the path if those directories do not already exist.

Column Name	Type	Description
skey	uint	A unique serial number assigned to each directory entry
location	uint	A reference to the location in which this directory exists
pathname	varchar(4096)	This is the relative pathname of the directory, relative to the "location" in which the associated files are stored
pathhash	char(40)	This is an SHA-1 hash of the pathname, which is used to identify all files having the same name
owner	varchar(256)	The name of the owner (converted from the numeric UID)
group	varchar(256)	The name of the group (converted from the numeric GUD)
mode	char(4)	This is a textual representation of the directory permissions. It is an octal number where the first digit indicates the SUID(4)/SGID(2)/sticky(1) bits, the second octet is the owner permission for read(4), write(2), execute(1), the third octet is the group permissions, and the last octet is the "other" permissions. For example, "0755" means that the file has owner read/write/traversal, group read/traversal, and other read/traversal permissions.
timestamp	uint	Timestamp when this row was last updated, which should be each time the discover process executes

9 Amazon Identity and Access Management (IAM) Policy

Pug needs to have the ability to read, write, and delete objects in an S3 bucket. Further, it needs to have the ability to list objects in an S3 bucket relative to the prefix. The following policy object would be sufficient for a bucket named “MyBucket” and a prefix value “pug”.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject"
            ],
            "Resource": "arn:aws:s3:::MyBucket/pug/*"
        },
        {
            "Effect": "Allow",
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::MyBucket",
            "Condition": {
                "StringLike": {
                    "s3:prefix": [
                        "pug/*"
                    ]
                }
            }
        }
    ]
}
```