



HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

**Software-Defined Storage for Big Data
on top of the OpenStack platform**

D3.2 SDS Toolkit initial prototype

Due date of deliverable: 31/12/2016
Actual submission date: 31/12/2016

Start date of project: 01-01-2015

Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.9
Number of pages	40
WP/Task related to this document	WP3 /T3.2
WP/Task responsible	MPSTOR
Leader	William Oppermann
Technical Manager	Michael Breen (MPSTOR)
Quality Manager	Raúl Gracia Tinedo (URV)
Author(s)	William Opperman (MPSTOR), David Coffey (MPSTOR), Michael Breen (MPSTOR) Ramon Nou (BSC)
Partner(s) Contributing	MPSTOR, BSC
Document ID	IOStack_D3.2_Public.pdf
Abstract	Public release of Block storage IOStack prototypes. Complete specification and APIs of the Block storage SDS toolkit. First description and evaluation of results obtained from validation in use cases using different standard benchmark tools to simulate standard workloads
Keywords	IOStack, Block Storage, Software Defined Block Storage, Konnector.

History of changes

Version	Date	Author		Summary of changes
1.0	27-10-2016	Ramon (BSC)	Nou	BSC Filter Description, evaluation and initial commit
1.1	16-11-2016	Ramon (BSC)	Nou	Arctur tests
1.2	02-12-2016	William Oppermann (MPSTOR)		initial commit
1.3	14-12-2016	William Oppermann (MPSTOR)		review commit
1.4	19-12-2016	Ramon (BSC)	Nou	requested 1st review revision
1.5	20-12-2016	Ramon (BSC)	Nou	requested 2nd review revision
1.6	23-12-2016	William Oppermann (MPSTOR)		requested 2nd review revision
1.7	29-12-2016	Ramon (BSC)	Nou	General document improvements
1.8	31-01-2016	Ramon (BSC)	Nou	requested 3rd review revision
1.9	31-01-2016	William Oppermann (MPSTOR)		requested 3rd review revision

Table of Contents

1	Executive summary	1
2	Introduction	3
3	SDS for block storage	4
3.1	Features and benefits of the SDS toolkit	4
4	SDS Gateway and Vendor plugin operation	5
4.1	OpenStack Horizon dashboard extensions	6
5	Konnector operation	7
5.1	Filter Functions	7
5.2	Konnector design	10
5.3	Filter Implementation	10
5.4	Filter toolkit	11
5.5	Standard Toolkit Filters	12
6	Advanced Filter description	13
6.1	Output Compress Filter	13
6.2	Compress Cache Filter	15
6.3	Deduplicated Cache Filter	15
7	Evaluation	16
7.1	Environment	16
7.1.1	Arctur environment description	16
7.1.2	Controlled environment description	16
7.2	SDS toolkit functional test results	16
7.3	Filter framework and performance results	17
7.3.1	Sequential and Random RD test 8K and 128K	19
7.3.2	Sequential WR test 8K and 128K	19
7.3.3	Random WR test 8K and 128K	19
7.3.4	CPU usage for point A and point B	20
7.3.5	Bandwidth sharing using the BW filter	21
7.3.6	Completion latency testing	22
7.4	Advanced Filters Evaluation	23
7.4.1	Output compress filter	23
7.4.2	Compress Cache Filter	24
7.4.3	Deduplicated Cache Filter	25
7.4.4	Filters analysis at Arctur testbed	25
8	Discussion and future development	27
9	Conclusions	30
10	Appendix	32
10.1	Konnector API	32
10.1.1	iSCSI initiator commands	32
10.2	Terminology	35
10.3	FIO test output	37
10.4	BitRev Filter example	38

1 Executive summary

The exploitation of data analytics has many challenges. One of these challenges is managing data storage back-ends for data analytics workloads. The traditional approach in building Big Data systems is to physically merge compute and storage on the same hardware node, this approach provides excellent coupling and performance between compute and storage but is very inflexible compared to a virtualised approach. The IOStack method is to use virtualisation, in particular the use of the open source OpenStack framework with extensions to allow easy menu driven deployment of data analytics jobs. This virtualisation of the analytics framework for various workloads shown in fig 1 creates a problem in how to provide storage for these workload clusters. Data analytics workloads can have many constraints, such as the amount of working storage required, the amount of compute required and the time to complete a given job. Meeting these constraints requires provisioning of the correct storage capacity at a given performance level. The IOStack project seeks to solve this problem with an SDS toolkit. The SDS toolkit is a set of stand alone storage tools as well as an integration with OpenStack to allow easy automated provisioning of data storage capacity at defined performance levels. Data analytic workloads can use three underlying storage types;

- block storage
- file storage
- object storage

Block and File storage are characterised as high cost per terabyte solutions whilst object storage is lower cost per terabyte. Block and File solutions have higher performance compared to object storage when measured in IO per second (IOPS) and megabytes per second of bandwidth (BWPS). Normally the higher IOPS of block solutions means the IOPS cost is lower for block storage than IOPS cost for object storage. This may seem counter intuitive as there are several open source object storage solutions however their IOPS performance is low when amortized over the hardware cost.

There is no easy rule in choosing block storage over object over file storage, the choice depends on the workloads and the performance, redundancy and resiliency that is required by the user. What is important is that the data-centre administrator has a choice of storage back ends so that he can choose and use the most appropriate storage for his workload. By choice we mean block or object storage, hard disk or solid state disk devices, choice of fabric and protocol, such as Fibre Channel or 1G, 10G or 100G Ethernet and what processing should be applied to his storage volumes.

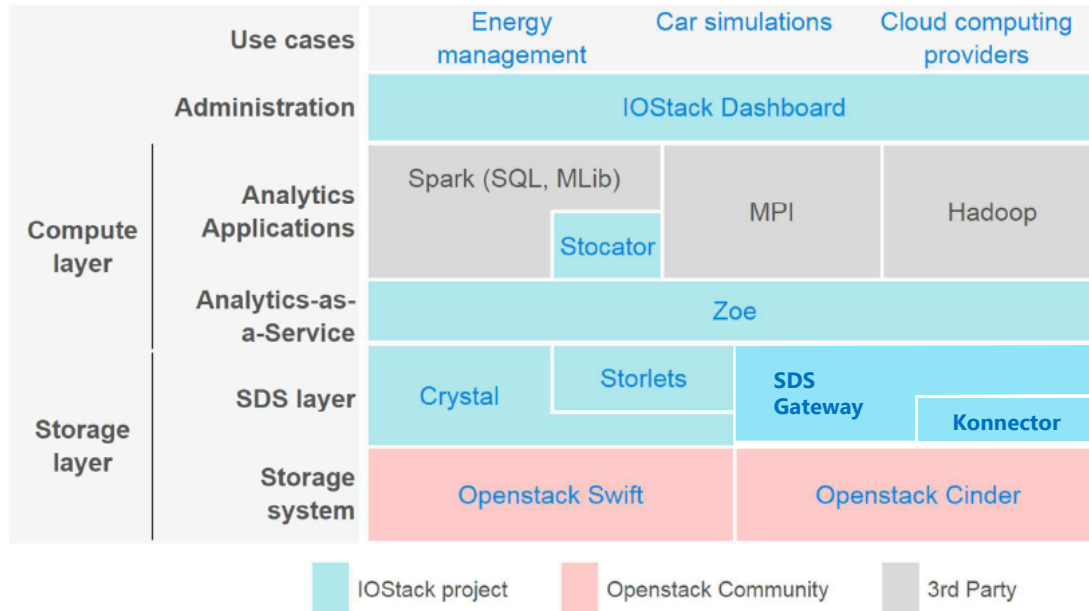
The SDS toolkit developed for WP3.2 provides an IOStack administrator the capability to create user defined storage services for specific analytics workloads. The SDS toolkit allows a user using a self service dashboard to provision block storage attach it to compute nodes, create an in-band stack of storage processing operations (known as storage filter functions) on top of a storage array volume and insert these filter functions between the storage consumer, in this case an OpenStack virtual machine and the storage array volume.

Block storage arrays are normally proprietary closed systems and complex, they are perceived as expensive and are delivered with a fixed set of storage features. Being mission critical systems they are slow to evolve and are inflexible innovation platforms. In contrast to proprietary block storage arrays, standard high performance x86 server nodes used for compute or object storage have an open software architecture and are very flexible platforms for innovation.

The IOStack SDS toolkit architecture leverage's the compute node open platform flexibility by moving storage functions (filters) into the compute node. These storage filters provide a means to create innovative block storage functions in-band on the compute node for the data analytics process.

The SDS toolkit has been implemented in the Arctur data center and has been successfully used to demonstrate automated block storage service provisioning as well as the development, test and deployment of in-band filter processing functions on data flows. The SDS toolkit is a working prototype with a list of planned enhancements.

This report presents the D3.2 work package (SDS tool kit) which includes a description of the block storage environment, a functional description of the SDS tool kit, the results of testing the SDS toolkit and several implementations of block storage filter functions.



46

Figure 1: Iostack Big Data components

2 Introduction

A trend in data centres is the constant growth of data and the processing of this data using data analytics to extract financial value. Storing and processing of data creates a provisioning challenge for the analytics user defining and running a data analytics job. The user when configuring his job is required to choose how many virtual machines he will use, how many cores and how much memory he will use as well as how much storage capacity and storage performance required. When the user is not the data centre administrator this is impossible without virtualised infrastructure using self service dashboards. Provisioning the resources required for the user should be automated without any intervention of the data centre administrator.

In IOStack provisioning of these resources is done using a high level self-service dashboard. The SDS toolkit provides the means for the user to choose at a high level storage capacity from a storage group and the means for the data centre administrator to create these storage groups with policies which define and enforce storage performance properties.

Provisioning requests from OpenStack are forwarded to the SDS Gateway which automates the provisioning of storage according to a configured policy so that the user gets the appropriate storage for his workload.

Policy based provisioning allows the SDS tools to automate provisioning with very fine grained storage properties, properties which are specifically required for a specific user workload. In Fig 10 we see how this fine grained provisioning is implemented. A storage group encapsulates which storage array devices should be used, the SAN type (ex. 40, 10, 1G eth or Fibre Channel 16G or 8G), media tier (ex. HDD, SSD) and which filter functions on the compute node should be inserted in-band in the data flow between the Virtual machine and the storage node. A chapter is provided for those who unfamiliar with storage terminology.

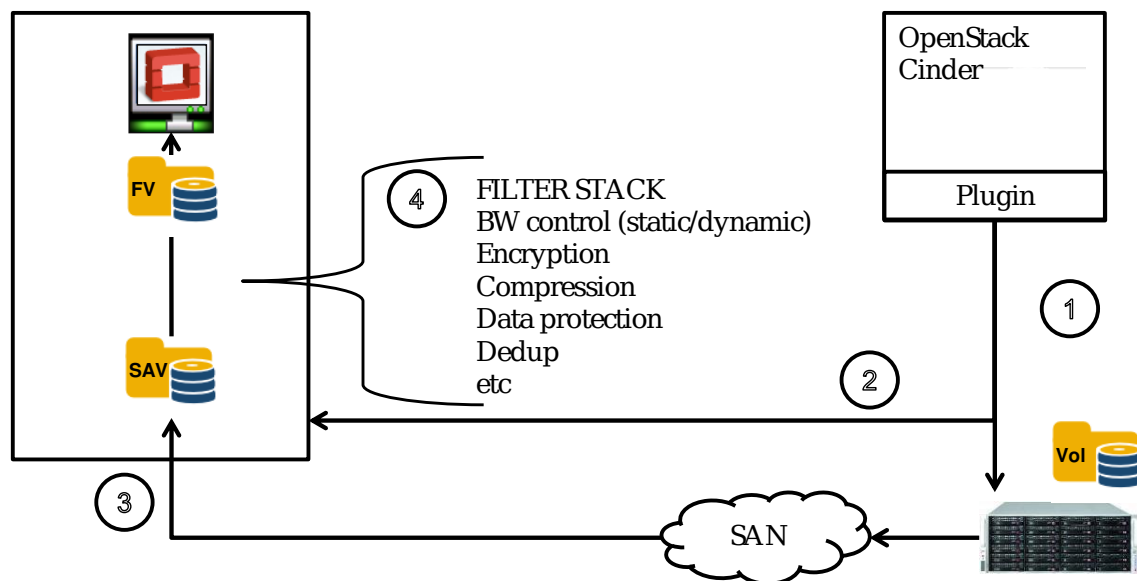


Figure 2: SDS Operation Overview

OpenStack is a general purpose cloud management software which can run data analytic workloads. OpenStack has a pluggable and extensible architecture, including extensions for data analytics.

The SDS toolkit manages block storage arrays in OpenStack allowing a user choice of different block storage media tiers, fabrics and protocols and importantly a means to extend block storage functionality by providing an in-band data filter stack on an OpenStack compute node. A filter stack is a layer of software on a compute node that is in-band between the storage array volume and the consumer on the compute node of the storage volume. In the data analytic environment the consumer is a Virtual machine (VM) running some stage of the data analytic process. A filter stack

can provide functionality to improve the data analytic function such as encryption, compression, deduplication or any other real time in-band data processing function on the data flow between the consumer (VM) and the storage array. One use case for the consortium partner Idiada was analysed and a filter implemented for this use case.

3 SDS for block storage

The SDS toolkit has three main components;

- An extension to the OpenStack management dashboard to allow SDS specific configurations and operations
- An SDS gateway which receives storage provisioning requests from the OpenStack dashboard and processes these requests
- A compute based module called Konnector which implements the storage filter framework. The storage filter framework is independent of the filter functions. Filter functions are software functions inserted in the data flow between the consumer such as virtual machine and the storage array volume. The filter framework is agnostic to filter implementation as long as the filter implements a set of standard entry points. A set of filters can be instantiated on demand by the filter framework as filters are implemented as Linux dynamic linked libraries files (.so files).

The Konnector project is stored in a public git repository <https://github.com/MPSTOR/Konnector>
The project website can be found at <https://mpstor.github.io/Konnector/>

The SDS toolkit in an OpenStack environment, shown in fig 2 allows a user to provision block storage through the openstack Horizon user dashboard according to a user defined policy (1), attach it to compute node (3), create a set of filters based on the user policy on top of the storage array volume (2) and attach a filter volume (4) to the consumer, in this case an OpenStack virtual machine.

In this deliverable, we also spent efforts validating the SDS framework for block storage. In particular, our validation methodology comprised of two sets of tests:

- functional testing of the SDS extensions to the management tools of OpenStack, in particular the Horizon user dashboard
- performance testing of the filter framework

Functional testing of the SDS toolkit within an OpenStack environment allowed the creation of storage groups with user defined policies. Policies provided the user with a means to configure the filter stack configuration per storage group. Volumes were provisioned from storage nodes and were successfully attached to compute nodes across a wide range of filter stacks. A filter stack is a set of inline storage operators inserted in the data flow between the storage provider (the storage Array) and the storage consumer ex. a virtual machine on a compute node.

Performance testing allowed the measurement of the overhead of the filter framework and the performance of individual filters. The overhead of the filter framework was measured by comparing a number of performance metrics at the top and bottom of the filter stack. Measuring performance at the bottom of the filter stack measures the raw storage array performance. Measuring performance at the top of the filter stack measures the loss or extra overhead of the inline filters.

3.1 Features and benefits of the SDS toolkit

The SDS toolkit provides a number of benefits in managing storage arrays and storage volumes with the filters;

SDS abstraction allows user choice of virtualised storage resources The SDS extension for the Horizon dashboard allows a user to choose not only capacity but also the storage type. A user can therefore choose whichever storage type that best suits his workload, an example could be a Gold Volume type which uses storage arrays with Solid State media tier, over a 10G SAN with a compression filter on the consumer node.

SDS abstraction allows data centre service creation through the creation of storage groups, data tiers and policies The SDS toolkit allows the datacentre Administrator to create specific storage services for particular workloads. These storage services map to storage groups, media tiers and policies over a range of fabrics. By combining filters within the storage group different services can be created, for example layering compression and encryption filters can be presented as a particular service within the datacentre.

Better management of shared storage arrays A key metric in purchasing storage is the cost per IOPS (IO Per Second) of the storage. This metric is increasingly important as data centres transition to All Flash storage arrays. All Flash storage arrays provide high performance in terms of IOPS however they are expensive. It is therefore important to provision not just the amount of storage in terms of Giga/Tera bytes of storage but also the IOPS and MBPS (Megabytes/sec) of each volume provisioned by the user. The filter framework allows the administrator to set a policy for the IOPS and MBPS for each volume created in a storage group. The user can then select a volume of X Gigabytes at an IOPS=100 or 1000 or 10000 depending on whether the storage array can provision this level of IOPS. A typical all flash array can provide up to 2Million 8K Random IOPS. In it therefore critical that the user has the tools to provision the IOPS and MBPS that his workload requires. In a multi-tenant environment a low priority workload may well use more IOPS and MBPS than the user intends for this workload, this "bad tenant" can therefore starve a workload that does require a specific amount of IOPS and MBPS for the workload to run within a specific amount of time. The SDS toolkit for OpenStack provisions storage from a storage group, each group has a policy that defines which set of storage arrays the storage volumes are provisioned and also the filters used on the consumer node. These filters can set the IOPS or MBPS of the provisioned volume allowing the user fine grain control of the MBPS and IOPS used by each workload.

Predictable behaviour In a multi-tenant environment by allocating not only the amount of storage but also the IOPS and MBPS of the storage for each workload leads to predictable behaviour. Testing has shown that the available IOPS of a storage can be arbitrarily carved up between different workloads. If the IOPS and MBPS of the storage are known this leads to predictable behaviour irrespective of what parallel workloads are being processed.

Lower management costs The SDS toolkit extension for the OpenStack dashboard allows the user to choose a volume type, this volume type is tagged to the compute group within which is configured a set of storage arrays, storage tiers, fabrics and consumer node filter functions. When a volume is created within this storage group the SDS Gateway virtualises and automates for the user all the operations of choosing which storage array, fabric and media tier to use. This greatly simplifies the process of provisioning storage. The second step of attaching a storage array volume and creating on the consumer node a filter stack is also a fully automated process. These complex provisioning tasks reduce the cost of provisioning managed storage by removing the need of the user to know anything about the datacentre internals and simply use the services created by the the datacentre administrator.

4 SDS Gateway and Vendor plugin operation

The SDS Toolkit has been implemented in the configuration shown in fig 3. The configuration is composed of the following components;

- (1) OpenStack Horizon plugins for Block Storage using the REST API plugin providing a Block storage SDS extension to OpenStack
- (2) SDS Gateway
- (3) SDS controller
- (4) Storage pool equipment over multiple SANs
- (5) Consumer node SAN termination points

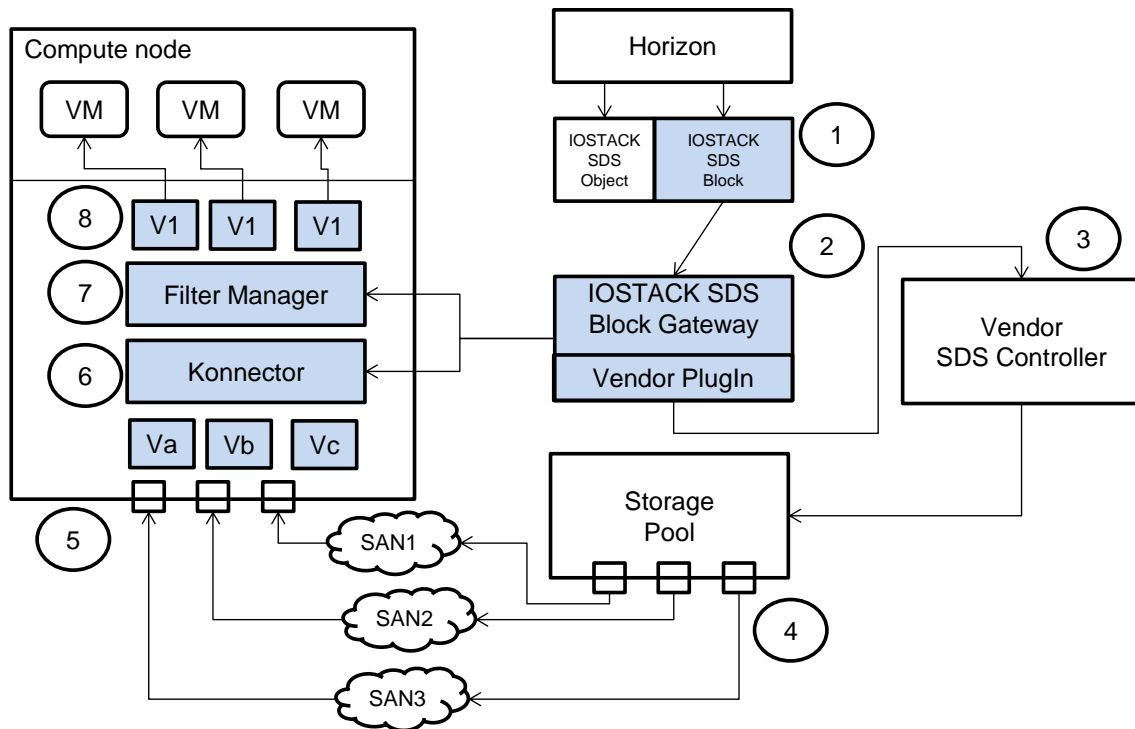


Figure 3: SDS Toolkit Overview

- (6) Konnector storage terminator of storage volumes
- (7) Filter Manager
- (8) Filter Volumes on top of storage volumes

The SDS Gateway provides a REST API for a client to manage storage. In fig 3 the client is the OpenStack Horizon dashboard issuing storage provisioning requests. These requests are CINDER API commands, CINDER is the OpenStack storage API used by the OpenStack Horizon dashboard. These CINDER API commands are issued to the SDS Gateway which translates them into requests understood by the backend storage arrays. This translation takes place in the Vendor plugin layer of the SDS gateway.

The SDS Gateway maintains an object model shown in fig 5 which is configured by the Horizon Dashboard SDS extension. A CINDER API request to the SDS Gateway will provision a storage volume from a specific storage group. The storage group can be created by the SDS extension of the Horizon dashboard using the SDS Gateway API. The SDS Gateway API provides a set of SDS functions such as creation of storage groups. A storage group has a name, a set of storage nodes and a policy. The group name is used by the user to select what type of storage is required for the workload (example Gold, Silver, Bronze). The storage provisioned will be from one of the nodes in the storage group. The policy configures default storage options for all volumes provisioned within that storage group. Some of the options are storage node specific, such as the fabric to export the volume on, other options are specific to the consumer node (Compute node) where the storage volume is attached to. The compute node specific part of the storage group policy defines the filter stack that is created when the storage array volumes is attached to the compute node.

4.1 OpenStack Horizon dashboard extensions

The SDS extension to OpenStack provides the user with a set of configuration and monitoring options. The configuration options allows the user to create storage groups which contain storage nodes, each storage group has one policy. The policy defines which media tier to use, default fabric

and filter stack to create when a volume is attached to the compute node. The filter stack is instantiated only when a volume is attached to a compute node. The Horizon extensions are used to create the configuration model shown below. This model is then used by the SDS Gateway and Konnector modules to instantiate and configure the filter stacks and volumes.

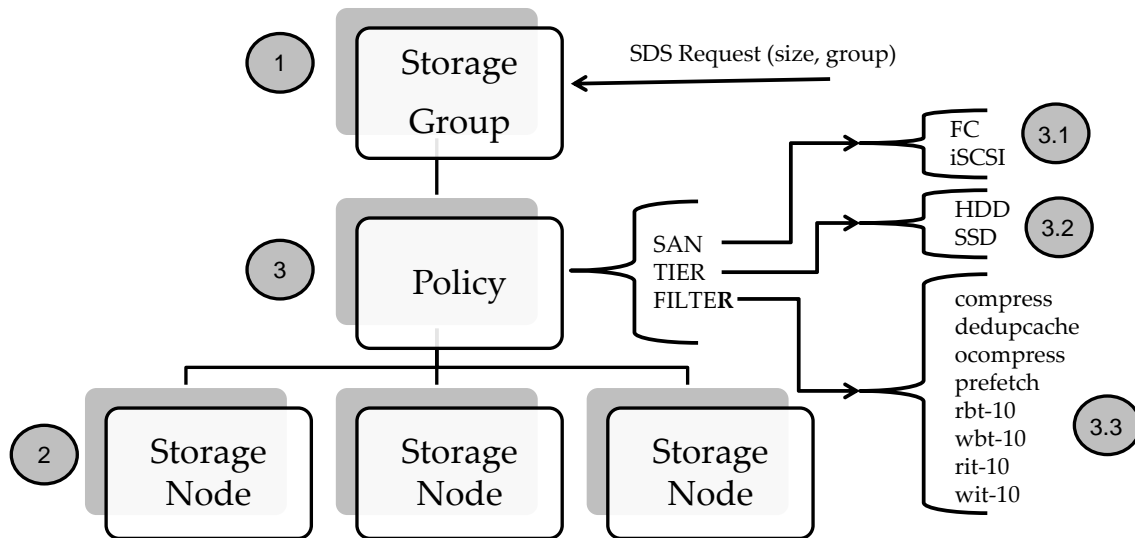


Figure 4: SDS groups and policy

5 Konnector operation

The consumer node (compute node) specific component defined in the storage group policy is a description of the filter stack that is dynamically built once the storage volume has been attached to the consumer node. Once a storage volume has been attached to a consumer node, the SDS Gateway shown in fig 5 uses the Konnector API to dynamically create a filter stack between the attached volume and the volume presented to the final consumer, ex. a Virtual Machine. This operation is shown in fig 3 in the steps 5, 6, 7 and 8. This schema allows storage volumes to be created on storage nodes and a stack of dynamic storage filter functions to be applied to the storage volume independent of the storage array volume and the storage node hosting that volume. For example a volume could be created on a storage node but the data to and from that volume could be compressed by a compute node filter.

The goal of the Konnector filter stack is to provide a flexible platform for innovation and value added functions on top of the storage array volumes.

Storage volumes are natively managed in OpenStack compute nodes by Kernel based drivers. Inserting filter functions on a data flow in Linux Kernel space would be extremely difficult. The Konnector framework moves the data flow from the kernel space into the userland space where filters can be dynamically created at run time. Run time creation of the filter stack is mandatory because the volume and its stack are only instantiated when the storage array volume is attached to the consumer node/consumer VM.

5.1 Filter Functions

Filter though a stand-alone function are an integral part of Konnector package. Filter functions are managed by the Konnector API. In theory any number of filter functions can be created, this is key to unlocking innovation of top of data flows. In Table 5.1a we present a list of examples is shown and the motivation behind developing these filters.

Compression Reduce the memory used for cache, compressing blocks. Increases the effective cache space in compressible content, but non-compressible content may suffer penalties. Other filters may

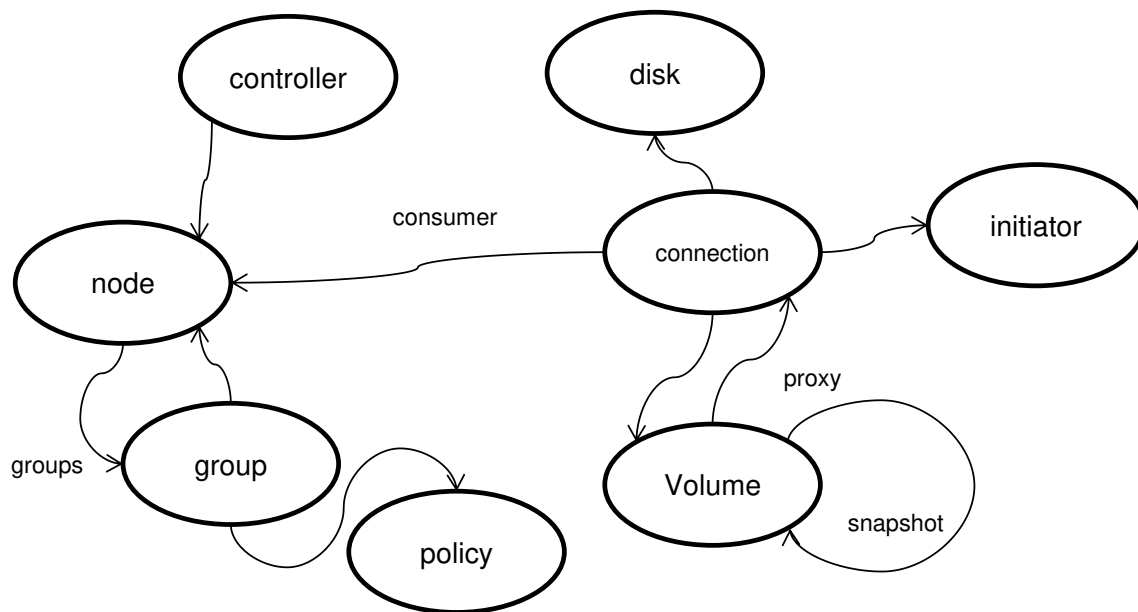


Figure 5: SDS gateway model

Table 5.1a: Candidate Storage Functions to be implemented as Filters

Filter Function	suitability	CPU load	IO Latency	BW impact
Compression	medium	high	medium	medium
Encryption	high	medium	medium	medium
Deduplication	medium	high	high	high
xN way Sync copy	high	medium	medium	medium
xN way Async copy	high	medium	low	low
Backup	high	medium	low	low
IO control	high	low	low	low
BW control	high	low	low	low
Block to Object	high	medium	low	low
Sample filters	High	low	low	low

generate compressed output, so the benefits come from a better space usage, however the main challenge is to export this to the user as the device needs to be expanded with a fixed ratio as it dynamic volume expansion is not allowed at this level. Compression is a cpu intensive algorithm and therefore will add latency to any operations either compressing or uncompressing the data. The more BW the more the compression algorithm will have to process and this will negatively impact the bandwidth. In some cases data will compress and uncompress easily (ex all zeros) in these special cases performance will increase.

Encryption Encrypted data flows provide two key advantages;

- Deletion of data can be achieved by simply removing the key, this reduces the deletion time of data from hours to seconds.
- Data is protected in a multi-tenant environment from data centre personnel and data centre tenants.

A Key management system is required to manage Keys securely under the user's control. Encryption is a cpu intensive algorithm and therefore will add latency to any operations. The more BW the more the encryption algorithm will have to process and this will negatively impact the bandwidth.

De-duplication De-duplication removes multiple copies of data stored in a the storage array. De-duplication reduces consumed space as only one copy of every unique record is stored, this can improve performance on de-duplicated data writes. A record of say 8Kbytes of data is known by an SHA value, if records have the same SHA the data is not written twice. De-duplication is a complex function, requiring a large high speed working storage space. De-duplication is also very cpu intensive in calculating SHA values for every record on the compute node. However, deduplication can be used also to increase the memory available for cache in a non-persistent way. The filter stores the last used blocks into memory (with a red black tree) indexed by the memory content itself. Deduplication is a very CPU intensive application requiring sha calculations and read modify write operations of the basic records, this adds considerable latency delay and lowers bandwidth.

Sync copy Providing multiple secure copies of data improves data resiliency. Data resiliency is one of the key attractions of Object storage. By providing filter based data resiliency low cost single controller block storage devices with high performance can be used in place of costly high availability enterprise class storage systems. A considerable amount of high speed meta-data is required to keep the status of the written blocks, the implementation of recovery procedures when a failed write IO occurs are complex. Copy is not a cpu intensive application as IOs are simply forwarded to multiple destinations. Sync copy requires that all writes be completed before a write is completed, this constraint increases the latency and lowers bandwidth.

Async copy Providing multiple secure copies of data improves data resiliency. Asynchronous data resiliency is one of the key attractions of Object storage. By providing filter based asynchronous data resiliency, a better compromise of performance and data resiliency could be achieved than by using Object storage. This is especially true if low cost single controller block storage devices are used. A considerable amount of high speed metadata is required to keep the status of the written blocks is required. Since the data is copied asynchronously recovery procedures to failed IOs are simpler than Sync copy to implement. Copy is not a cpu intensive application as IOs are simply forwarded to multiple destinations. Async copy does not requires that all writes be completed before a write is completed, the lack of this constraint means latency and BW are not negatively impacted.

Backup A filter that marks block ranges as dirty when written to a storage volume could then read and copy these block ranges to a backup store. Backup is means of achieving additional data resiliency and copies of storage volumes at a point in time. A point in time backup requires a snapshot of the volume, this is a complex operation. Backup also requires that the volume is in a flushed state from the host OS, i.e no cached data is in the compute system, everything has been flushed from the filesystem to the storage volume. This is actually a simpler task to implement for a filtered volume than a non filtered volume as filtered volumes have local knowledge of the compute node file system. Backup is not a cpu intensive application as IOs are copied from a source volume to a remote when required with little data processing. Backup is essentially a scheduled activity so does not interfere with the normal IO operation and has little impact on latency or bandwidth.

IOPS control IOPS control is important as it allows a user to provision not only terabytes of data from a storage array but also a portion of its IO capability. IOPS control is completely compute node based and requires no API interface to the storage array controller. Controlling IOPS on storage arrays with high IOPS rates does not work in badly behaved multi-tenant systems. A badly behaved tenant is one which does not obey any IOPS rate limiting. For IOPS provisioning to be affective all volumes must subscribe to an IOPS rate control mechanism.

BWPS control BWPS control is similar to IOPS control except the read and write megabytes per second (BWPS) is controlled. BWPS faces the same challenges as IOPS control.

Block to object For applications requiring block semantics but Object storage class speed and cost, a filter to make the block to object transformation would be very useful, examples of such tools are S3backer and CEPHs RDB driver. Porting a tool like S3Backer to the Filter Framework.

Sample filters Sample filters are useful templates for anyone developing a more complex filter. The sample filter should be a good example of how to code a more comprehensive filter.

5.2 Konnector design

The Filter stack is dynamically created as storage array volumes are attached to a compute node, this is made possible as each filter is implemented as a .SO (a dynamic linked library). The Filter manager when requested through the Konnector API will dynamically build the filter stack using a set of .SO dll library files. This .SO approach allows the filter stack to be built on demand, it also allows any third party to create filter functions and add a data inline processing function into the data flow between the consumer (ex. Virtual Machine VM) and the storage Array volume. The storage array volume is agnostic to the filter function since the storage array just sees data in/out of the attached storage array volume on the consumer node, it does not see nor is aware of any of the filter transformations.

The Konnector block diagram is shown in fig 6, the principal components of the system are the flow of SAN SCSI commands (1) into the LINUX LIO core, the backing store for this SCSI device is shown as /dev/sdx (2). This device /dev/sdx (2) is the storage array volume created by the SDS gateway, exported from the storage array and imported in the consumer node. In non IOStack Open-Stack this volume would simply be attached to a virtual machine. In the IOStack enabled compute node the data flow (SCSI Read and Write Commands) are written through two buffers (3-4) into userland space. In userland space the filter manager (5) recovers the data and routes the data to the stacked up filters configured for that volume (7). Data from one filter can be routed into one filter then to succeeding filters (6) in the stack.

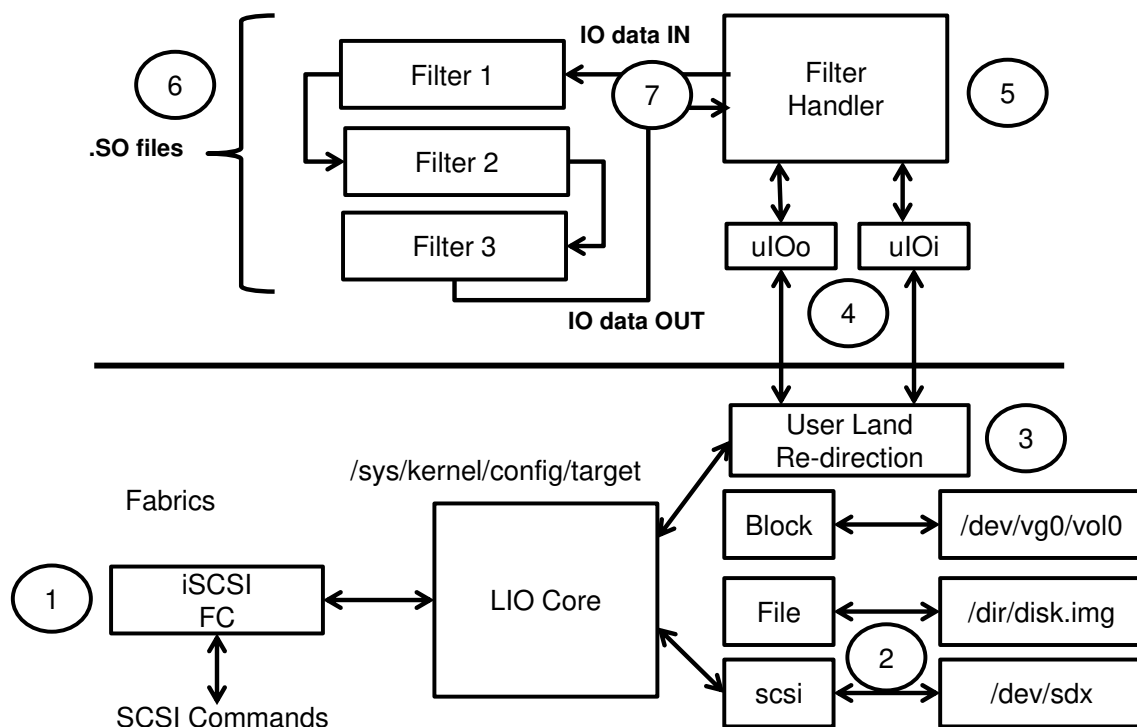


Figure 6: Konnector Block Diagram

5.3 Filter Implementation

The filter objects are built from C source files. The SDS toolkit provides a number of skeleton filter samples such as 'nop' which serves to act as a template for filter development. Within this filter, there are five functions which are run-time linked to the filter manager for version filter manager 1.1.

- write transformation

- read transformation
- get-name
- pass-args
- pre_read

The write transformation - void write-xform(void* buf, unsigned long cnt, unsigned long offset, bool *doWrite) is passed a void pointer to the payload data together with a length argument 'cnt' and an argument 'offset'. Essentially, this function exists to perform a transformation at its defined filter level on payload write data bound for the device. The last parameter is an output parameter intended to avoid writing the content to the disk, as it is already done in the filter. For example, filters that write to other zones of the device will bypass the final write.

The read transformation - void read-xform(void* buf, unsigned long cnt, unsigned long offset) uses the same prototype definition, is called at the same predefined filter execution level and is designed to act on the read payload data from the device en-route to the initiator.

get-name - which can be called from the manager. This just returns the name of the filter which is defined as a static string in the filter object. This is just added for debug purposes and is called when the filter daemon is executed in debug mode.

pass-args - This function allows the user to pass arguments to the filter function when the filter is instantiated. The arguments can be specified using the syntax of the filter specification in the Horizon dashboard, ex myfilter arg1=0x10, arg2=helloWorld. The Arguments are filter specific and are not parsed by the filter manager, they are passed to the filter transparently.

pre_read - int pre_read(void* buf, unsigned long cnt, unsigned long int offset, unsigned int fdesc, bool * doRead). This function that is called before the read-xform by the filter framework offers a way to the filter to bypass the read function that goes to the non-filtered device. It is intended for cache filters, so if the content is already in memory there is no need to go to the non-filtered device and then the doRead parameter will be 'false' and the content requested will be already in the buf pointer.

5.4 Filter toolkit

The filter framework provides a development toolkit that allows easy creation of filters.

The current SDK provides a build directory which compiles all filters and copies to the filter directory where the filter framework attempts to match the API call to create a filter volume with the name of an existing filter. A filter is instantiated through the Konnector API, the API is passed the name of the filter which must be present in the filter directory. The filter name to be instantiated and its arguments are stored in the storage group policy created by the OpenStack Horizon dashboard. Every storage policy has a property which is the list of filters to be instantiated when a storage volume is attached to a compute node.

Each filter must code the four entry points described in the previous section, section 5.3.

The filter framework and the filters are entirely decoupled. A filter could be compiled anywhere and simply copied to the to Konnector filter directory.

The listing below shows how the filter files are compiled and stored in the Konnector-filters directory. If a Konnector API call to start a filter with name XYZ is called, a search for the filter with name "XYZ" is made in the directory "Konnector-filters" and if found the filter is deployed in the filter stack for the designated volume. Filters can be stacked one on top of another, the data flow will flow from the top of the stack to bottom for "data writes" and from the bottom to the top for "data reads". A sample filter BitRev is shown in section 10.4 BitRev Filter example. In Section 10.4 the source, header and make tools for the filter are shown. The example shows how the required entry points can be coded as described in section 5.3. To write a new filter the following steps are performed.

- step 1: code the filter entry points as shown in the example with the particular function you wish to implement, this is the coders decision.
- step 2: create a header file as shown in the example.
- step 3: build the filter using the script in the example or use the script below to build all the filters in filter directory.
- step 4: copy the filter .so file to the directory Konnector-filters

The filter manager will now be able to access and load the filter into a filter stack on top of a storage array source volume.

Script to build all filters in the default filter source directory.

```
for i in $( ls filters ); do
( cd filters/$i && gcc -Wall -fPIC -c "$i"_filter.c" \
  && gcc -shared -Wl,-soname,"lib_"$i".so.1" -o "$i".so" "$i"_filter.o" \
  && if [ -f "$i".so" ];
then
  cp -f "$i".so" /usr/lib64/Konnector-filters
else
  echo "File"$i"so"_"does_not_exist"."
fi )
done
```

5.5 Standard Toolkit Filters

The toolkit filters serve the purpose of test filters to verify the filter framework is working. They also serve as sample filters for people to develop their own filters. The IOPS and WBPS filters are part of the filter framework that provides IO and BW rate limiting.

The SDS toolkit provides a number of standard filters including;

The NOP filter provides no processing on data flow between the attached storage and the storage consumer. In this regard it can be seen as the overhead of the filter framework itself.

The XOR filter provides a simple obfuscation of the data written to the storage volume by performing an XOR operation on the data written to the storage and an XOR operation on all data read from the storage array. The XOR filter is a simple filter but is useful in that it requires the filter function to perform a transformation on all read and written data. To use the xor filter add the string "xor" in the SDS dashboard policy field.

The BitReverse filter provides a simple obfuscation of the data written to the storage volume by performing a bit reverse operation on the data written to the storage and an bit reverse operation on all data read from the storage array. The BitReverse filter is a simple filter but is useful in that it requires the filter function to perform a transformation on all read and written data. To use the BitReverse filter add the string "bitrev" in the SDS dashboard policy field.

The IOPS filter allows the user to set a value of the IOPS allowed to traverse the filter stack. The IOPS can be set for read IOPS, write IOPS or both. To use the IOPS filter add the string "rit 10, wit 10" in the SDS dashboard policy field for say 10 read IOPS and 10 write IOPS.

The BW filter allows the user to set a value of the BWPS allowed to traverse the filter stack. The BWPS can be set for read IOs, write IOs or both. To use the MBPS filter add the string "rbt 10, wbt 10" in the SDS dashboard policy field for say 10 megabytes per second read and 10 10 megabytes per second writes.

6 Advanced Filter description

The filters developed here go a step further of the original behaviour of the filter framework. The original behaviour was a 1:1 block transformation (reading or writing), so for each read or write request to the non-filtered device the same request is done to the filtered device. For our scenarios we need to be able to do a $n:n$ transformation, so we need to do additional reads or writes to the non-filtered or filtered device. For example, prefetching or cache filters need to check before the real read if the content is available or not, then the framework knows if the real read should be done or not. On the other hand, output compress filter generates a new filesystem to the real device transparently, so before it writes the real data it should know if the data should be written or not (doWrite parameter of the write-xform call).

The filters developed at BSC are classified in four types:

1. Prefetch filters (prefetch)
2. Cache filters (dedupcache and compress)
3. Output modification filters (OCompress)
4. Evaluation filters (mockup)

Prefetch filters preload data in advance to the filter to avoid the network latency. prefetch filter is divided in two filters, the first one logs the offset and sizes of the data that is being used in the VM. The second filter preloads the data in advance. We are going to develop a new filter on the next period that will enable Just-in-time prefetching so the blocks are only preloaded just when they are going to be used. Prefetching will allow important latency reduction on storage as the devices are not directly attached to the client and are provided through a network.

Cache filters stores any block read into the filter. The filter objective is to increase the performance on workloads reusing data. The cache is reduced using deduplication (dedup) and compression (compress), so we can store more data with less memory usage and surpass the buffer cache space. The deduplication method was used on a FP7 project called IOLanes [1], implemented directly in the kernel.

Output modification filters generates a different output from the one expected. The main difference is that the filter is persistent, so the output can only be read if the same filter is used. ocompress generates a compressed file system using two compressors (for Idiada use case). The filter is transparent for the user. However, further modifications in the filter framework are needed to allow to export a, i.e., 6 GB real volume, and present it inside the VM as a 10 GB volume. The main issue is that the increment of space should be static and in advance (using some % gathered or introduced by the user) and can not be changed as it will confuse the VM operating system. Idiada needs this filter due to that their data is highly compressible (but heterogeneous) and they are using a lot of space. Idiada is using actually a in-home special compressor that provides better space reductions than other available compressors, but with the use of the filter they will be able to access the compressed data transparently.

Evaluation filters tries to introduce several parameters to evaluate the framework, it can also be used to introduce latency delays or CPU usage delays in the filter workflow. mockup filter has those parameters, so delays are introduced on each read or write request.

The advanced filters are available at github.com/bsc-ssrg/BlockStorageFilters-IOSTACK.

We are going to describe the major filters developed on the next subsections.

6.1 Output Compress Filter

The filter creates a compressed filesystem inside the unfiltered device and present a normal device to the user. The compressed filesystem was created for the Idiada use case due to that they have

big output files that are highly compressible. The compression is done at first by minilzop [2], that provides a high performance compressor but when the block is not compressible it tries to use zlib [3] that allows a slower but better compression.

Other compressors can be used and implemented, along with other techniques integrated at block level, for example if the block content is of type "CONFIDENTIAL" use encryption as we can see on the next example code. Being able to use encryption is important for the use case of Idiada due to they are using sensible data that should be protected.

```
void write-xform( void* buf, unsigned long cnt,
                 unsigned long offset, bool *doWrite )
{
    int class = analyse (buf, cnt);
    int newSize = 0;

    switch (class)
    {
        case CONFIDENTIAL:
            encrypt (buf, cnt, key);
            *doWrite = false;
            break;
        case NONCONFIDENTIAL:
            newSize = compress (buf, cnt, key);
            *doWrite = false;
            break;
    }

    /* We store the block on the filter ,
    as we are creating an output filesystem.
    i.e., We need to store metadata to know which content is stored.
    */
    storeModifiedblock (buf, cnt, offset, newSize, class);
}
```

The filter uses the `pre_read` operation (so we can redirect reads to another zone of the disk) and the `doWrite` output parameter (so we can redirect writes) of the filter framework. To support compression, we designed a filesystem that compresses per block and stores in its metadata the type of compression used in order to support different compressors. The structure of the metadata structure can be seen in the Table 6.1a, *DISKRECORDS* and *DIRTYBLOCKS* should be setup in advance or pass them as parameters so they fit the volume. One important point of the OCompress filter is that it needs support of the *tcmu-filter* to export to the VM a volume which size is a the original size plus a % of the intended compression ratio. As the device must be static the size can not be dynamic and dependent of the content, this modification is still not implemented.

Table 6.1a: Metadata of the compressed filesystem

Name	Description	Size (bytes)
MAGICCONST	Identify Filter version	8
LASTBLOCK	Last Block Used	8
DIRTY	Last Dirty Block Used	8
RECORD*DISKRECORDS	offset, size and type of a block storing data	11
RECORD*DIRTYBLOCKS	offset, size and type of a block storing unused blocks	11

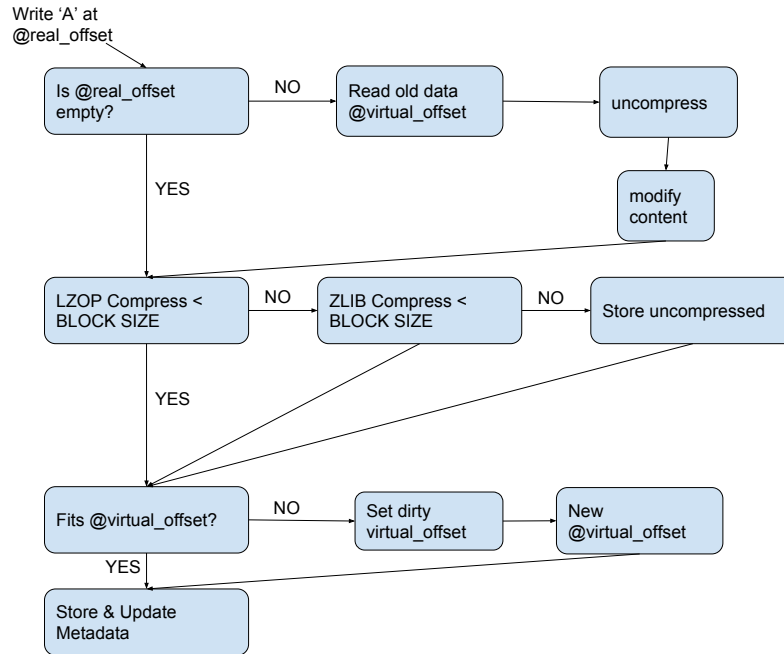


Figure 7: Simplified write workflow on the Output compressor filter

The filter workflow can be found in Figure 7, the figure describes a write workflow, that is the most costly operation. It also includes the read workflow, as writes can be unaligned and 1-byte based, therefore we need to read the old block, uncompress it, modify it and finally compress it again and store it.

This filter is persistent, so once it is used the volume can not be accessed without the filter.

6.2 Compress Cache Filter

The compress cache filter (and the deduplication filter using the same concept but with deduplication instead of compression) is a filter that shows benefits when the data cached is reused. As this is not the case on the different use cases, we are going to evaluate it using the Idiada dataset and forcing a reuse. The initial Idiada dataset is truncated to 8GB, and we are going to give 5GB of memory to the filter, enough to keep the full compressed dataset on memory and show the benefits comparing it to the same setup without the filter. Without the filter, the memory available for the buffer cache is potentially of 8GB. However, it will not fit due to other processes laying on main memory. The filter uses the `pre_read` function to disable the real read if the filter decides it. Compression is done with `lzop` [2] and `zlib` [3], but any compressor works. The structure used to store the cache in the filter is a red-black tree indexed by offset having as contents an integer to store the compressed size and a buffer with the compressed content¹.

6.3 Deduplicated Cache Filter

Similar to the previous filter, but using deduplication and with a 100% deduplicable dataset (10 GB) we will obtain better results as the cost of compression is removed. The duplicated filter stores the blocks in a red-black tree having as key the block content, and in another red-black tree, indexed by the offset, the pointer to the content on the first tree². This technique avoids the hash of the content and the needed test of collision, the performance is similar because `memcmp` shortcuts on the first difference. On the deduplicated cache filter, we implemented eviction of blocks using different techniques (random removal, first block, ...). Testing shows very subtle changes on the performance

¹The memory overhead is 16 bytes per block as minimum, depending on the rb tree implementation

²The memory overhead is 24 bytes per new block and 12 bytes per deduplicated block as minimum, depending on the rb tree implementation

between the different methods..

7 Evaluation

7.1 Environment

We had run the evaluation on two environments, the first one is using the Arctur testbed to evaluate the filter framework, the second one is using a less powerful server and a VM to evaluate the filters in a memory and CPU controlled environment while maintaining the datasets with a controllable size (to evaluate, for example, caching filters).

7.1.1 Arctur environment description

The Arctur environment used for the evaluation has a storage array with several devices setup in a RAID which are served to a node which contains the filter framework (24 cores Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, with 12GB of memory). The filter framework node is where the evaluation is done.

The Arctur nodes are shown in Fig 8 which consists of two storage Arrays, the SDS Gateway, a compute nodes and the cloud controller. The SDS Gateway for reasons of simplicity of setup was installed on the storage node .52. The SDS Gateway is a stand-alone service which could be installed on a separate node. Two storage nodes were used to demonstrate different hardware in different storage groups. Storage node .52 had two storage tiers one on HDD (Hard Disk Drives) and one on SSD (Solid State Disk Drives). Solid State disk drives given their low latency allowed testing of RANDOM IO patterns in the test VMs and Compute node. A single high performance multicore compute node was used to run test VMs (Virtual Machines) and test the Filter Framework. All nodes were connected on the management network to the cloud controller. The compute was connected to the SAN network. In our configuration for simplicity we chose the management and SAN network on the same network range.

7.1.2 Controlled environment description

The controlled environment is using direct attached devices to the VM running the filter framework. This allows to test directly the overhead of the different filters, and allows us to control the memory offered to the filter so we can check cache policies. This also avoids possible network bottlenecks.

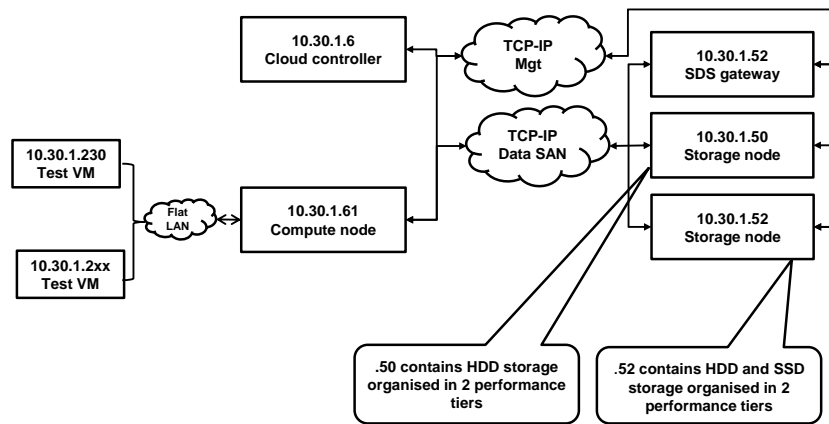
The VM is setup to run 4 GB of memory and 2 cores. The devices used are a 256 GB SSD (SAMSUNG SSD PM851 mSATA) and for some filters we also use a slower device (USB with higher latency) to show more clearly the benefits. The overhead of the filter framework can be extracted of the comparison of the use of the direct native device and the framework with a NOP filter. However, a more exhaustive evaluation has been done in Arctur testbed.

The host system is an Intel(R) Core(TM) i7-4700 CPU with 16 GB of memory. We always clean the different caches of the system on each repetition of the tests.

7.2 SDS toolkit functional test results

Table 7.2a below shows the functional test results. As demonstrated in the EU commission review all the functional components of the SDS toolkit are demonstrable and functional. The SDS toolkit has been deployed in Arctur and is available as a test and development platform for BSC and MPSTOR. The functional testing required configurations of storage groups with associated policies. Fig 4, Fig 10, Fig 11 below shows the setup steps.

1. A storage group is created (ex. Gold storage)
2. Physical storage nodes (by ip address) are associated with the storage group
3. A policy is created which specifies the SAN type, the media tier and a set of filters that will be deployed when the storage volume is attached to the compute node. The filter definition is a string of filter names and arguments for example in the filter field the following **xor, rit-10, wit-10, nop** would instantiate a stack of "xor" filter with a read IOPS of 10 IO per second, a write IOPS of 10 IO per second and finally the "nop" filter.



39

Figure 8: Arctur Overview

7.3 Filter framework and performance results

The performance test was designed to measure the efficiency of the filter framework with a nominal filter by measuring performance at Point B (Storage Volume) and Point A (Filter Volume) as shown in fig 9. The FIO test tool was executed on the Compute node in order to test the filter framework. Testing point A from a VM was not the methodology used as this would create noise over the KVM para-virtualised disk drivers between the Linux Kernel and the Virtual Machines.

The Filter chosen was a simple encrypt Filter which on writes XORed every byte with 0xAA and wrote the new data to the Storage Volume, on reads the inverse operation was performed.

A test tool was used to run IOs to the filter volume at the two test points, Test point A and Test point B as shown in fig 9.

Test point B was used to measure the raw performance of the storage array volume.

Test point A was used to measure the performance of the filter volume across the filter stack to

Table 7.2a: Functional testing results

Test	result	comment
Create storage policy with defined filter	pass	Uses SDS dashboard
Create storage policy with defined media tier	pass	Uses SDS dashboard
Create storage groups with defined set of storage nodes	pass	Uses SDS dashboard
Provision attach storage volumes from storage groups to compute nodes	pass	Uses SDS dashboard
Instantiate filter volumes on compute node	pass	uses Konnector
Attach filter volume to VN	pass	uses Konnector
IOPS Filter	pass	uses FIO and VM
BWPS Filter	pass	uses FIO and VM
Volume metrics	pass	uses metrics dashboard 9

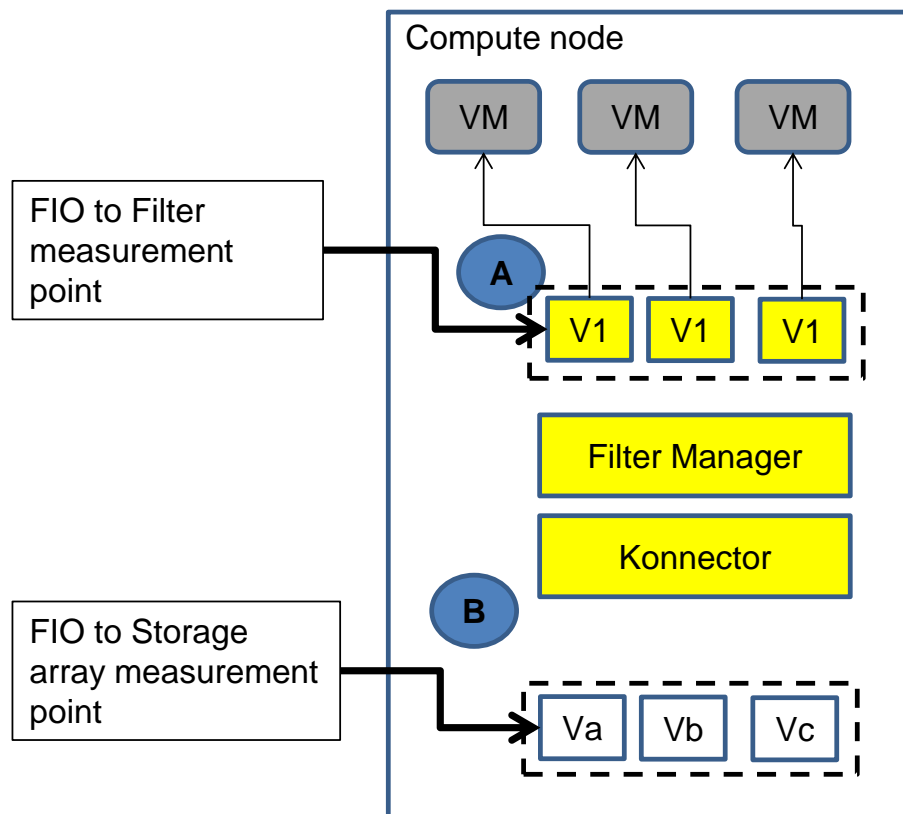


Figure 9: Konnector Performance Test measuring points

the storage array volume.

Prior to running any tests it was important to drop the LINUX caches to make sure no caching affects disrupted the measurement. To free pagecache, dentries and inodes the following command was used.

```
echo 3 > /proc/sys/vm/drop_caches
```

The test tool used was FIO which wrote a variable number of drives using Read/Write and 8K and 128K buffer sizes.

FIO produces a test report for each test configuration (see appendix A), in addition to the FIO results additional metrics were collected using collectD, collectD is a tool that measures key metrics and outputs to a range of viewer tools and CSV files. In the test configuration results were collected as .CSV files. The FIO sample test report shown below includes the IOPS and BW/sec of the tested storage volumes as well as a number of latency measurements. Data from these reports are graphed and presented below.

A set of test configurations were created for RANDOM and SEQUENTIAL IO at 8K and 128K IO sizes and over a varying number of attached storage volumes 2, 5, 10, 15, 22 volumes.

The collectD tools instrumented the consumer compute node and collected a set of metrics in CSV format which include:

- cpu usage
- disks stats
- network stats
- memory
- processes

The screenshot shows the OpenStack Storage Policies management interface. A sidebar on the left contains navigation links: Project, Admin, Identity, SDS Controller (selected), Object Storage, and Block Storage. The main area is titled 'Storage Policies' and features a table of existing policies. Callouts provide examples for specific fields:

- Policy Name:** Points to the 'Name' column.
- SAN:** Points to the 'San name' column, with examples: 40G Eth, 10G Eth, FC 16G/8G.
- Media Tier:** Points to the 'Tier' column, with examples: Mission Critical, SSD, Archive.
- Filters:** Points to the 'Filters' column, with examples: wit 50, rit 70, compression.

	Id	Name	San name	IO/s read	IO/s write	MB/s read	MB/s write	Tier	Filters	Created at	Actions
	2	fast	Eth0	-	-	-	-	Mission Critical	prefetch	2016-04-19T10:57:32Z	Edit Policy
	3	slow	Eth0	-	-	-	-	Archive	xor, rbt-10	2016-04-19T10:58:58Z	Edit Policy
	17	policy2	-	-	-	-	-	Mission Critical	xor	2016-05-25T13:51:48Z	Edit Policy
	19	mb-pol-1	-	-	-	-	-	Mission Critical	nop	2016-05-27T14:28:06Z	Edit Policy
	20	mb-pol-2	-	-	-	-	-	Mission Critical	xor, rbt-10	2016-05-30T10:25:25Z	Edit Policy

Displaying 5 items

Figure 10: Storage policy

- irq
- load

7.3.1 Sequential and Random RD test 8K and 128K

The sequential and Random RD test of fig 12 for 8K and 128K buffers measured at point A shows that we achieve SAN line speed and saturate the data link. The back-end storage array media is solid state disks so the random and sequential performance data is very similar. The affect of smaller or larger data buffers is negligible on measured performance. Performance measured (not shown) at points A and B were identical and saturated the SAN link.

Conclusion The Filter Framework does not adversely affect the READ or WRITE MBPS of the Filter volume, i.e the filter volume is as fast as the storage volume.

7.3.2 Sequential WR test 8K and 128K

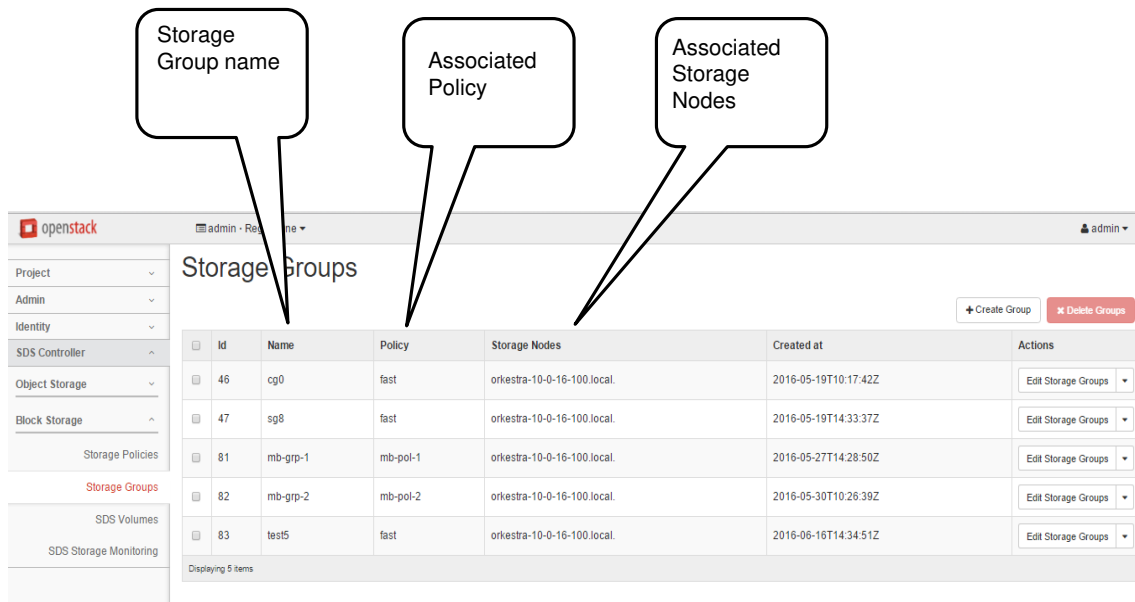
The sequential WR test of fig 13 for 8K and 128K buffers measured at point A shows that we achieve SAN line speed and saturate the data link. The back-end storage array media is solid state disks so the random and sequential performance data is very similar. The affect of smaller or larger data buffers is negligible after 5 disks on measured performance. Performance measured (not shown) at points A and B were identical.

Conclusion The Filter Framework does not adversely affect the sequential WRITE BW of the Filter volume, i.e the filter volume is as fast as the storage volume.

7.3.3 Random WR test 8K and 128K

The random WR test of fig 14 for 8K and 128K buffers measured at point A shows that we achieve SAN line speed and saturate the data link. The back-end storage array media is solid state disks so the random and sequential performance data is very similar. The affect of smaller or larger data buffers is negligible after 5 disks on measured performance. Performance measured (not shown) at points A and B were identical.

Conclusion The Filter Framework does not adversely affect the random WRITE BW of the Filter volume, i.e the filter volume is as fast as the storage volume.



The screenshot shows the OpenStack Storage Groups management interface. A table lists storage groups with columns for Id, Name, Policy, Storage Nodes, Created at, and Actions. Callouts point to specific columns: 'Storage Group name' points to the Name column, 'Associated Policy' points to the Policy column, and 'Associated Storage Nodes' points to the Storage Nodes column.

Id	Name	Policy	Storage Nodes	Created at	Actions
46	cg0	fast	orquestra-10-0-16-100.local.	2016-05-19T10:17:42Z	Edit Storage Groups
47	sg8	fast	orquestra-10-0-16-100.local.	2016-05-19T14:33:37Z	Edit Storage Groups
81	mb-grp-1	mb-pol-1	orquestra-10-0-16-100.local.	2016-05-27T14:28:50Z	Edit Storage Groups
82	mb-grp-2	mb-pol-2	orquestra-10-0-16-100.local.	2016-05-30T10:26:39Z	Edit Storage Groups
83	test5	fast	orquestra-10-0-16-100.local.	2016-06-16T14:34:51Z	Edit Storage Groups

Figure 11: Storage groups

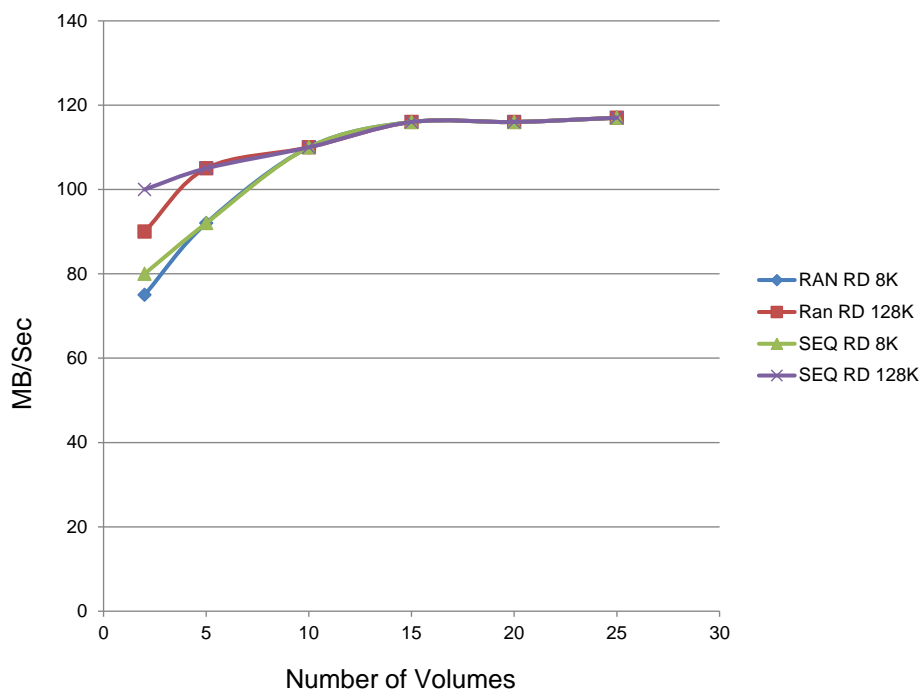


Figure 12: Sequential and Random RD test 8K and 128K

7.3.4 CPU usage for point A and point B

This test is the most important test in understanding the impact of the filter framework. The results are shown in fig 15. The node used to measure the cpu usage is a single processor 22 core system. There is no simple way to measure the CPU utilisation such as top as it does not capture the usage across all CPUs. Since the bandwidth achieved in both cases measured at point A and point B is the same we can simply measure the system CPU usage, the affect of FIO being common to both. Fig 15 is a normalised value across the 22 cores as a percentage of the total CPU. The CPU usage was calculated from the individual usage of the collectD cpu usage measurements for all 22 cores. In practice no more than 3 cores were ever active running threads for the filter stack. The measured

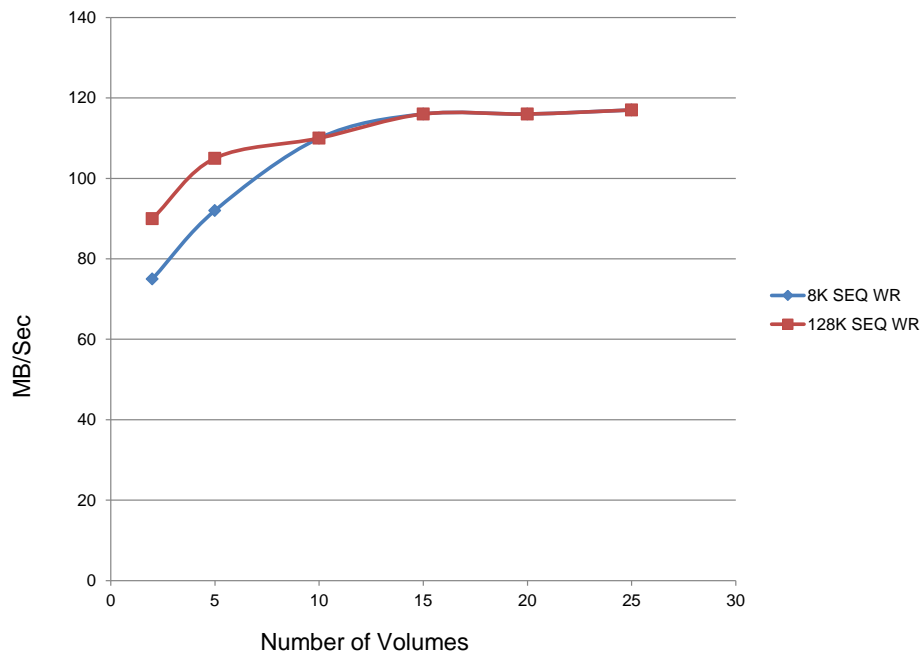


Figure 13: Sequential WR test 8K and 128K

results show approx 10 times more CPU usage at the top of the filter stack compared to the results at the bottom of the filter stack. It is important to note that the overall CPU usage of the system is quite low (10 percent) whilst driving the system at line speed over 22 attached storage volumes. The results also show us that the CPU usage reaches a plateau after 5-10 attached storage volumes. This corresponds to the point at which the line speed is saturated and adding more attached volumes only adds marginal CPU load to the system.

Conclusion The Filter Framework from previous results does not affect the storage performance however the CPU usage increases as more filter volumes are attached to the system. This result is understandable in that more attached volumes requires more CPU load to execute the filter function and execute the filter framework.

A control test of a NOP filter would have been useful to calculate the affect of the filter framework on its own without any inline data processing.

7.3.5 Bandwidth sharing using the BW filter

This test is designed to see how well a user can carve out available BW from the storage array. The results are shown in Fig 16. In this test a number of volumes were created and attached to control VMs. In each VM (UBUNTU VM) a single instance of FIO was run, a total of 6 VMs were run simultaneously and the BW measured in each VM, see fig 16. In order to avoid caching noise the caches were dropped on both the LINUX host system and the virtual machines prior to testing otherwise inconsistent results were obtained. The tests were run for 10 minutes to allow the system reach stability. The measured results were in very good agreement with the predicted results from the SDS toolkit filter settings over a range of BW settings. In this test only volumes with a configured BW filter were used. The use of non filtered volumes tended to use up to 50 percent of the available BW. This affect demonstrated a weakness in the filter management framework in that when using BW filters all storage volumes must be filtered as unfiltered volumes tend to go as "fast as possible" and use BW which should be used for the filtered volumes.

Conclusion The filter framework is an effective tool in provisioning not just storage but also BWPS and IOPS from the storage pool. Two conditions are required for this framework to be effective

- A) the total demand has to be less than the capacity of the storage pool

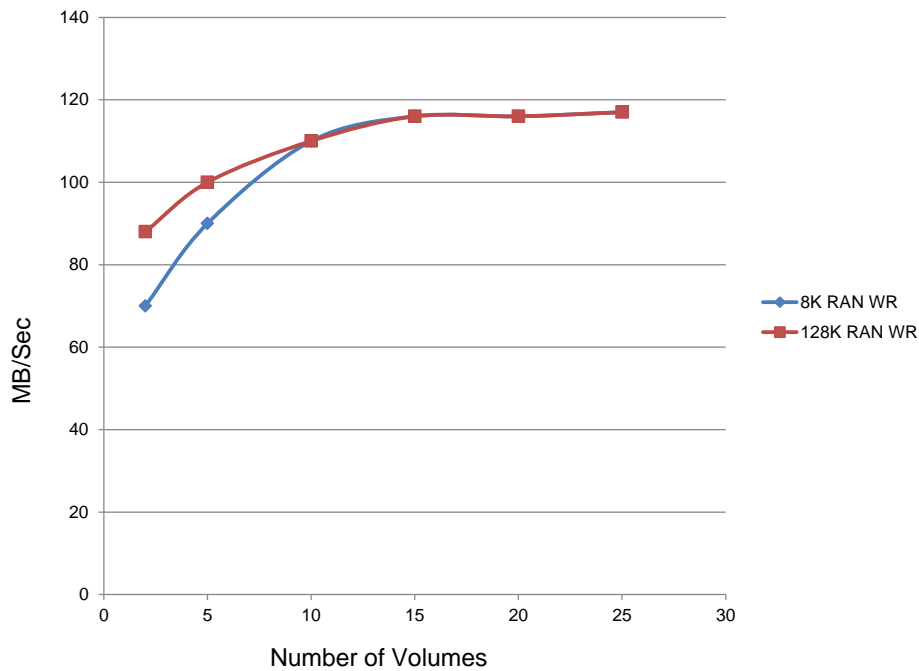


Figure 14: Random WR test 8K and 128K

B) only volumes with a BW or IOPS filter can be used as volumes without a BW or IOPS filter will go as fast as possible and drive the system into saturation

7.3.6 Completion latency testing

A metric provided by FIO is the completion latency which calculates the time it takes for each IO to complete. The IOs are then grouped into percentile buckets. With a lot of overlapped IOs will complete over a range of times. Figure 17 shows the completion latency for a system with 2 attached filter volumes and a system with 22 attached filter volumes. The completion time has been normalised since in both cases as the storage was running at line speed, i.e BW of 2 vols was approx equal to BW of 22 volumes, therefore the completion time of the 22 volumes system must be approx 10 times that of a two drive system. The results of this test are shown in fig 17.

Conclusion The result shows that the basic shape of the two drive system and the 22 drive system is the same. The filter framework as it scales does not introduce any bias or unfairness into the IO scheduling.

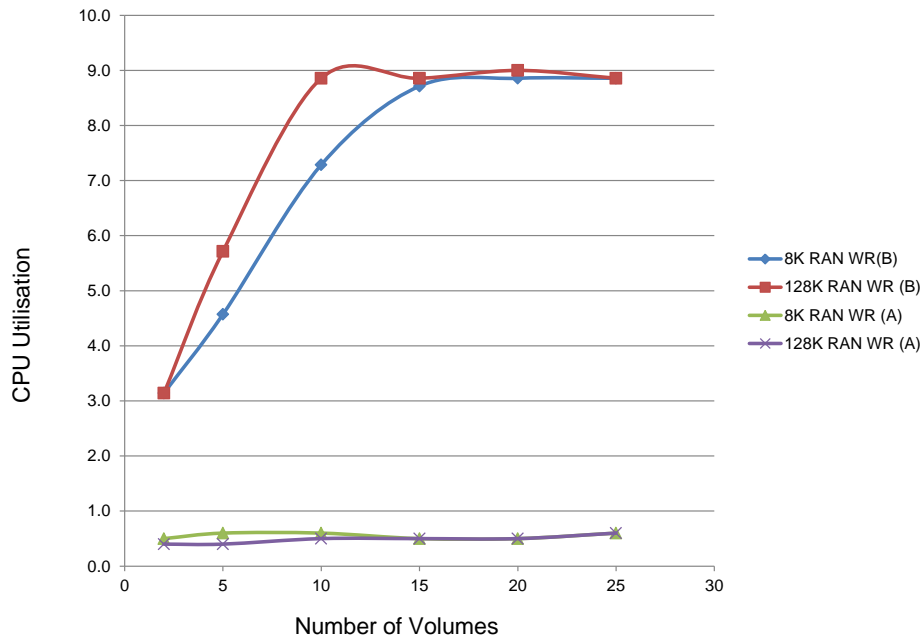


Figure 15: CPU usage for point A and point B

7.4 Advanced Filters Evaluation

We evaluate the filters using 10GB datasets with different content: The Idiada dataset, a 10 GB random generated file and a 100% compressible and deduplicable dataset. Only the Idiada dataset is copied from another device, as it can not be generated, producing lower transfer times, the other datasets are generated directly. The evaluation is done in the Controlled environment by default, but if they are done in the Arctur environment we will specify that on the figure captions (and the description in the text).

7.4.1 Output compress filter

We evaluate the filter using the output provided by Idiada, which presents a 60% of data reduction via compression and has a size of 10GB, we also use a random content file (which will try lzop and then zlib, so the overhead will be high) and a file with 1's (which avoids kernel and VM optimizations, but produces a high compressible content).

Files are already available inside the VM or are created in fly by lightweight processes. The filesystem is ext4 (the client needs to format the volume because the compression is transparent).

The compressed device via the OCompress filter, tries to compress using lzop. If the compressor can compress the block it is stored, if not, we try to use a zlib compressor (which is slow). This is an option of the compressor due to that the Idiada dataset can be compressed efficiently with an in-house compressor. However, due to confidentiality issues, we created the filter to allow multi-compressor techniques per block but does not use the real compressor.

The Idiada dataset offered, which is 10GB and can be reduced to 4GB with the most effective compressor, is stored to the compressed device. We present the normalized results in Figure 18 which shows similar normalized results for writes and reads using the filter. Despite that the absolute performance throught the filter is low on writes (40 MB/s - 50 MB/s depending on the content), the read performance goes from 200MB/s of the NOOP filter to 170MB/s going throught the OCompress filter as we need to read less data, however with the tested device we do not get performance benefits. Performance benefits will come when slower devices are used as we will show in the compress cache filter. The compressed dataset goes from 10GB to 5.7 GB. If we only use lzop, we reduce the data from 10GB to 6.3GB, and the performance increases as we do not have to check the other compressor. As we can notice comparing NOOP and Native results, the overhead is lower on writes as we get

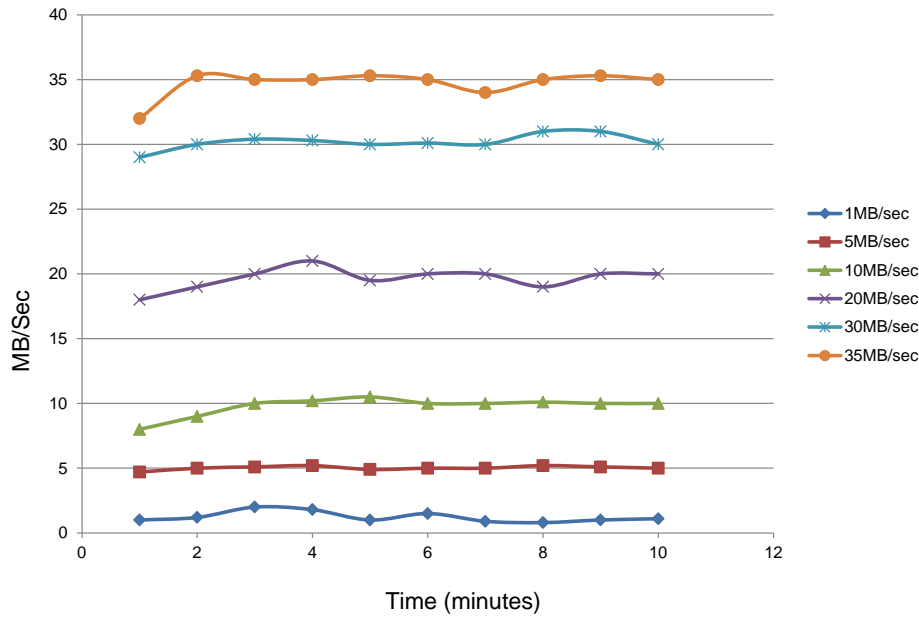


Figure 16: Bandwidth filter with multiple volumes

benefits from the buffer cache implementation on the kernel. Reads are sequential, and synchronous, so optimizations are not being used.

The worst behaviour, due to the double compression workflow, is obtained with a random dataset. The normalized performance can be seen in Figure 19. We can see that the NOOP and OCompress performance on READ is similar, this is due to that the data is stored uncompressed. However, the WRITE performance is very low as every blocks needs to be tested through the two compressors.

The best behaviour is when we use a 100% compressible dataset: The compressed disk is keep very small and the performance increases. In the Figure 20 we can observe as the WRITE performance surpasses the native and NOOP filter (due to that we are storing only compressed data and metadata to the device). READ performance is very high, more than the native device attached in the VM, as if we read 4K we bring to memory a lot of compressed blocks.

We have also the CPU behaviour of the different experiments, we can compare how NOOP and the OCompress filters behaves and how the OCompress is dependent of the content, as expected. Figure 21 shows the write CPU usage distribution using a violin plot for the NOOP filter, and the three compression scenarios. Figure 22 does the same when we are reading. We can see that reading is more lightweight than writing. And that when we are writing the content is important, for instance, writing random data spends near the 100% of CPU due to the double compression technique.

Compression will get benefits from devices with a bigger latency, as we will see on the next filter. Also, applying heuristics will provide speed improvements, for example avoiding the compression of random content.

On the other hand we could not replicate the experiments on the Arctur testbed because the double compression produces timeouts and the iSCSI interface fails putting the device offline.

7.4.2 Compress Cache Filter

The evaluation is done using two different devices, a fast SSD as the Output Compress Filter, and a high latency device where the benefits of a prefetch are more perceptible. As we are going to show on this subsection, SSD's speed is so high that using memory cache (and having to uncompress it) is not showing high benefits. Other techniques as deduplication or Just in Time prefetching will show benefits as we drop the compressor cost. In Figure 23 we show the normalized performance on the two devices. We removed the first read (that will be slower as it needs to go to the disk), such reads are presented on table 7.4a. With a fast device, compression has very small benefits (and it is not

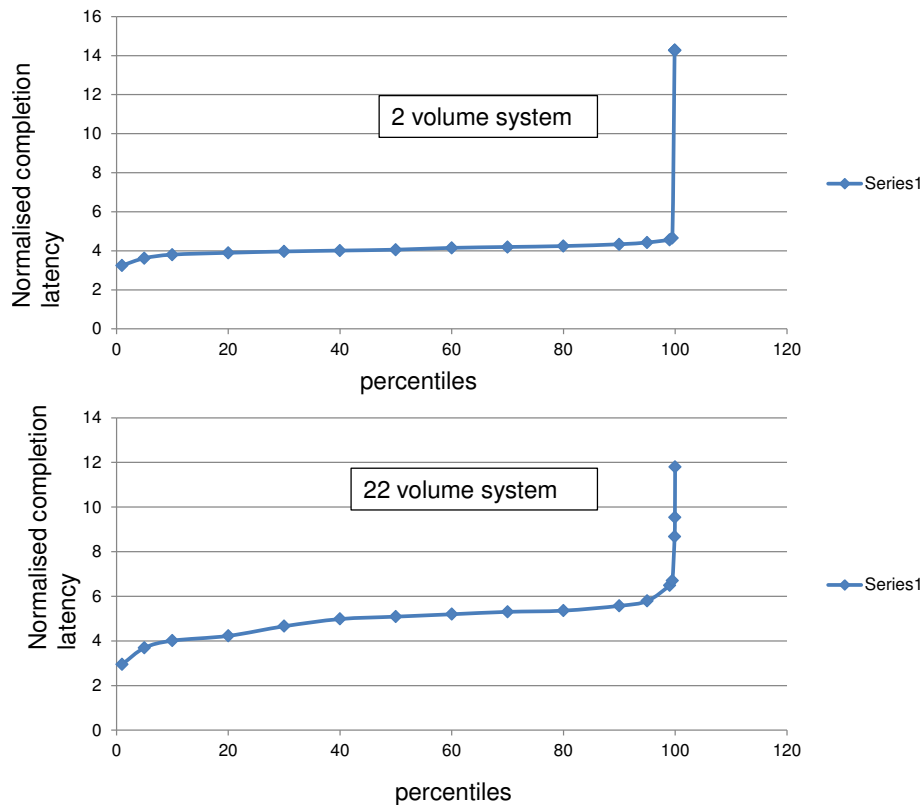


Figure 17: completion latency of 2 and 22 volumes

faster than reading the data directly), however with a slow device we increase the performance of the Native filter up to 3x (with the Idiada dataset).

Table 7.4a: First Read Time(s) using cache compress filter (and without filter)

Native Slow	NOOP Slow	Compress Slow	Native Fast	NOOP Fast	Compress Fast
95.66	108.33	114.09	38.24	30.01	63.96

As in the Arctur testbed the Idiada datasets fits the memory of the system we are unable to test it (as cleaning the caches would not be fair).

7.4.3 Deduplicated Cache Filter

In Figure 24 we show the normalized performance on the two devices. As the previous experiment, we removed the first read (that will be slower as it needs to go to the disk), such reads are presented on table 7.4b. As we can see, the results are similar to the compression cache filter, but in this case as the CPU cost is zero the performance improves with the fast device also.

Table 7.4b: First Read Time(s) using deduplicated cache filter (and without filter)

Native Slow	NOOP Slow	Dedup Slow	Native Fast	NOOP Fast	Dedup Fast
128.21	130.32	130.31	26.19	35.53	38.14

7.4.4 Filters analysis at Arctur testbed

As the write performance is slow, we can not test de OCompress filter with the two level compression due to timeouts. However, we provide a Dedup cache filter comparison as it does not use the write

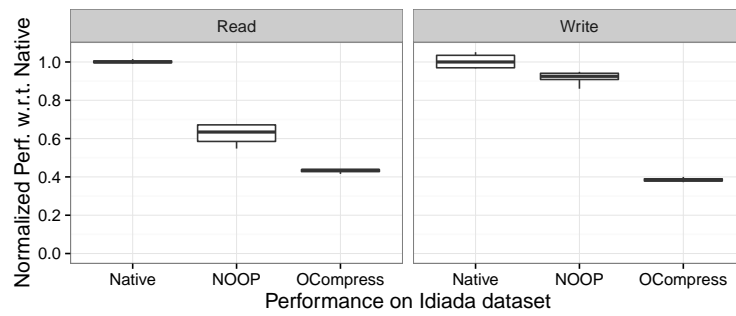


Figure 18: Idiada Performance with OCompress filter (Dual compressor)

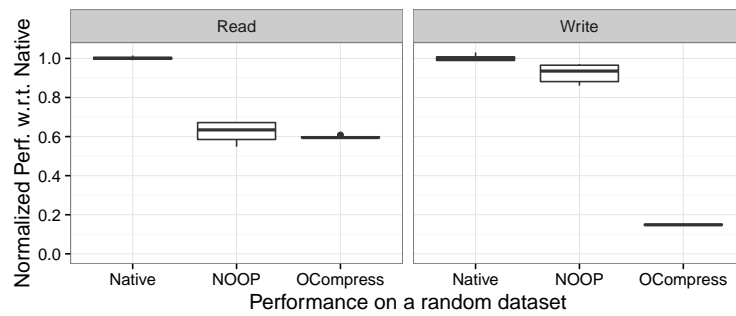


Figure 19: Random dataset Performance with OCompress filter (Dual compressor)

capabilities of the device.

With respect the controlled environment, we increased to 20GB the deduplicable dataset, so it will not fit in memory. We present the read results on the Figure 25. We are evaluating native and the filtered and cached device.

For comparison, the first read through the native device needs 179.71 seconds and through the deduplicated cache 183.09 seconds. We can observe how we get a similar performance to the one obtained on the controlled environment with the slow device, showing clearly the benefits of the technique on such datasets.

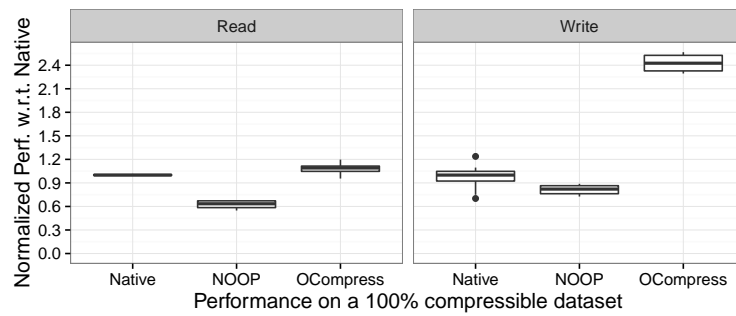


Figure 20: 100% compressible Performance with OCompress filter (Dual compressor)

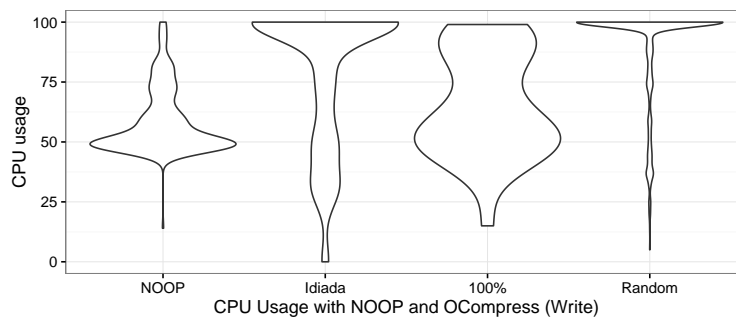


Figure 21: CPU Usage when writing

8 Discussion and future development

Issues raised during testing: A number of issues occurred which require follow on analysis; Results were measured by dropping both Virtual machine caches and host compute node caches. Without this results were inconsistent, in some cases GB/sec speeds were measured due to local caches. In a real life scenario caches are an important component of the system performance. Further testing is required with caching not dropped to see what is the performance impact.

The system does not work well if IOPS and BW filter volumes are mixed with volumes that have no BW or IOPS filter. A management strategy needs to be defined which allocates IOPS and BW to all volumes by default and rebalances those volumes with no set IOPS or BW filter if demand saturates a device in the end to end connection. This is a complex problem which requires a global view of the compute nodes, physical interfaces, switches and storage array capabilities.

The MPSTOR testing used FIO as the test tool and used only block IO testing. BSC used file system testing and in some cases writes returned an error. The cause of this error is not understood at this time.

The above issues should be completed in the next phase of work.

Additional development: Additional management capability is required in the filter management framework.

A) A management function in the SDS Gateway so that filtered and unfiltered volumes and filtered volumes with and without IOPS and BWPS filters can be managed. It seems there should be a single nexus for the management of all volumes otherwise bad and noisy tenants will cause unexpected behaviour.

B) An extension to the Konnector API to allow for uploading and registering of filters, this probably would require some signing and authentication of filter .so files.

C) A management function in the SDS Gateway to provide dynamic balancing of the storage volumes system wide, the BWPS and IOPS filter management should be managed in the Konnector API module and not in the Konnector run time framework. Moving IOPS and BWPS out of the filter

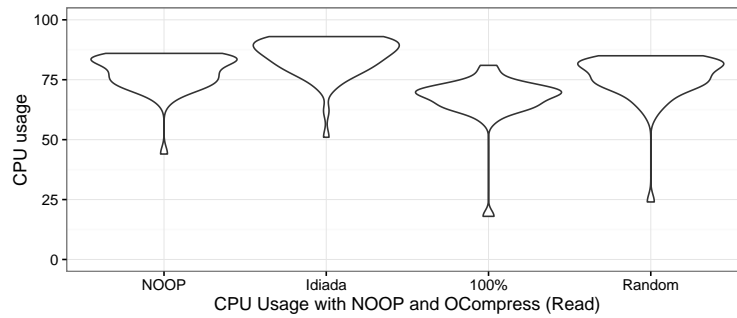


Figure 22: CPU Usage reading compressed data

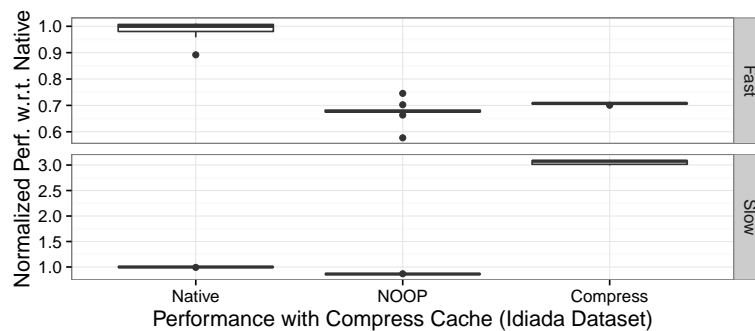


Figure 23: Normalized Performance obtained using compressed cache filter with the idiada dataset in different re-reads.

would allow real time dynamic balancing.

In fig 26 we show the end to end provisioning of each resource in chain between the consumer node and the provider node. In order to properly provision with an SLA that can be met, a model of the components between the end points should be created in the SDS Gateway. Provisioning of storage along the RED line in fig 26 indicates a carve out of the available resource. Provisioning of storage along the Green line in fig 26 indicates a carve out within the remaining available capacity. This remaining available capacity needs to be dynamically recalculated as new demands are made on the system as shown in figure 27.

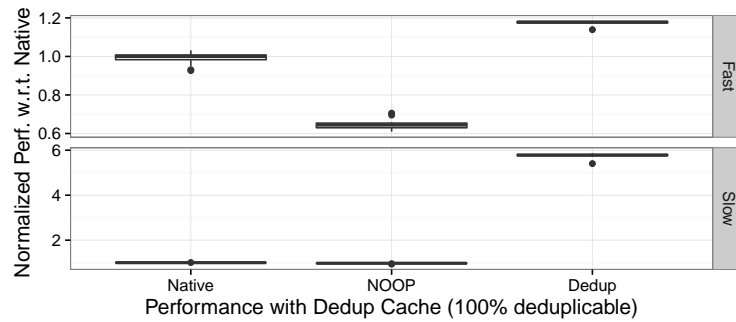


Figure 24: Normalized Performance obtained using compressed cache filter with the idiada dataset in different re-reads.

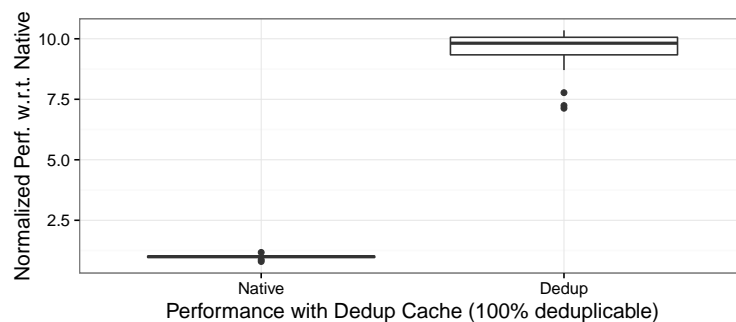


Figure 25: Normalized Read Performance obtained using dedup cache filter with a 20GB fully deduplicable dataset.

From the filter perspective, some work can be done at the filters to improve such performance, we have implemented on some of the filters (dedup, compress and ocompress) cache mechanisms and threading to decouple the execution from the sequential execution of the filter framework. On most scenarios this is a good option as the multithreaded code can be coded having in mind that no more than one request will enter the system. Some of the filters, improved the framework to allow filters that are far from the initial 1:1 transformation with operations like the `pre_read` and `pre_write`. One of the most advanced filters is the `OCompress` filters, but in order to be 100% functional there is a change that need to be done to the framework. The framework, if we use a compression filter, should present the filtered device (statically, as disk size can grow dynamically in a filter agnostic environment) with a $n\%$ more space. This increase should be definable based on past executions or the knowledge of the compressibility of the data.

Future plans for the advanced filters developed is to create a Just in Time prefetcher, that will log the reads of a device, build a Markov Chain to get the possible next blocks and bring them to memory before it is used. A similar filter will be created for Object Storage. We also plan to continue developing the `OCompress` filter for Idiada. Actually we will present an evaluation of the compressibility of their dataset to justify such filter, however the fact that the actual testbed does not allow to execute it (due to slow devices) adds some difficult our tasks.

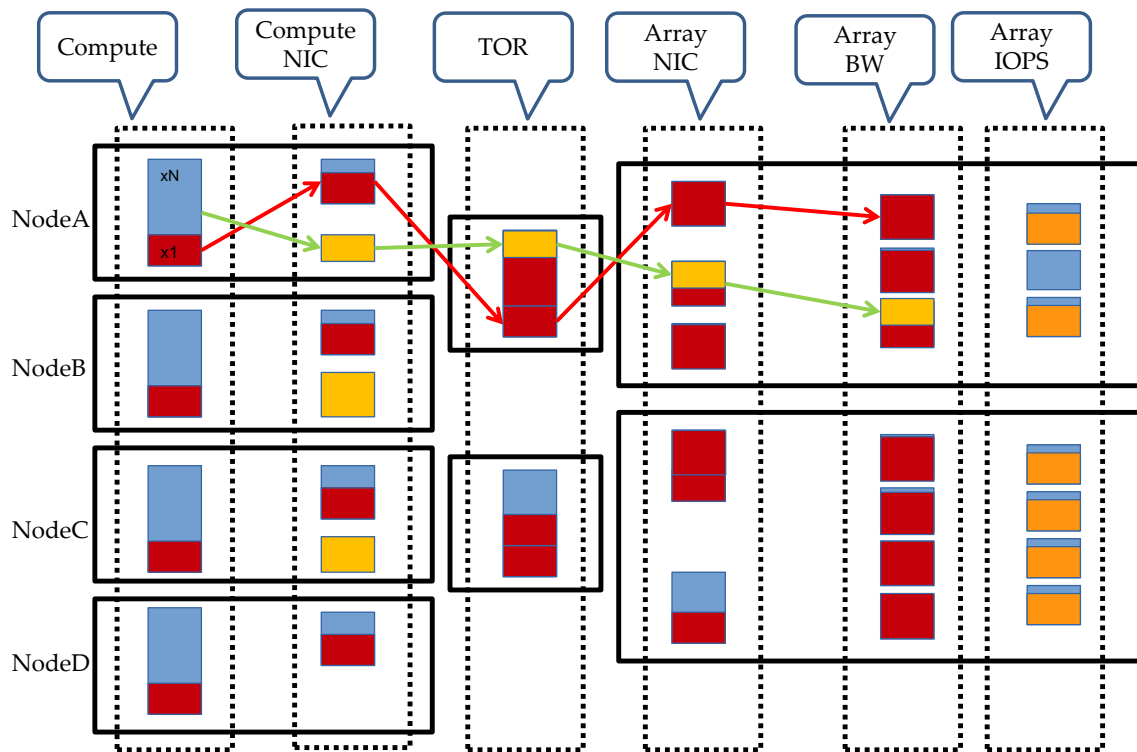


Figure 26: dynamic bandwidth management

9 Conclusions

The conclusion of the testing of the filter framework is that the SDS toolkit is an effective set of tools that meet the goals of the project. The goals achieved were:

Working prototype A working prototype has been developed which shows all the component parts functioning together.

User control of datacenter infrastructure The extensions to the Openstack horizon dashboard allows an administrator to create volume services using policy parameters so that the user can choose the correct type and capacity of storage for his work-load.

Automated provisioning The policies, storage groups and volume filter services configured in the admin dashboards are automatically deployed when volumes are created and instantiated, this means a high level of automation of complex configurations has been achieved. This allows the user to deploy the correct storage configuration easily for his workload.

Inline filters The Filter framework allows the creation and real time deployment of inline processing in the compute node of the dataflow between a VM and the storage Volume, this inline data flow processing was a core goal of the project.

Added value Inline filters Added value filters were developed that go beyond proving the feasibility of filter use. These added value filters improve the workload requirements of the application.

The toolkit provides the following features: A Horizon dashboard interface to create storage groups of storage nodes, media tiers and consumer node filters.

A flexible means to create storage policies and attach storage policies to storage groups.

By using storage policies storage volumes with defined IOPS or BW characteristics the valuable resource of storage array IOPS and BW can be managed correctly.

A filter framework that does not adversely affect the performance of the attached storage but does require extra CPU resources depending on the filter type being executed.

Future work: The SDS toolkit in this work package was tested with generic workloads representative of Big Data workloads. The next steps will run Big Data workloads with added value filters and

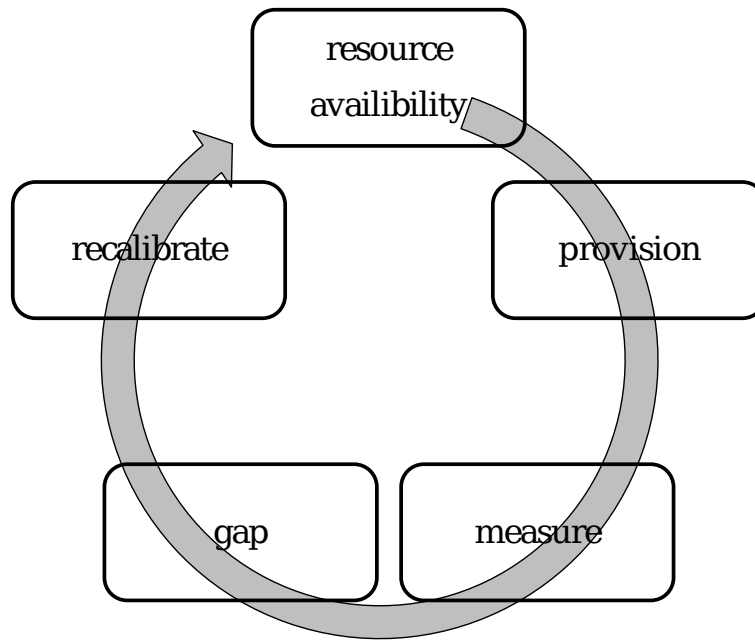


Figure 27: dynamic bandwidth management

demonstrate improvements of the Big Data applications.

A number of lessons have been learnt from the testing which will be used to direct the next phase of development.

10 Appendix

10.1 Konnector API

This section describes only the Konnector extensions to the targetd API. The functions below are accessed in the same way as described in the targetd API document using JSON RPC on TCP port 18700. For example, using curl, a create-filtered-volume command could be executed like this:

```
IP=192.168.2.61 or wherever the Konnector service is running curl -user admin:password -X POST http://IP:18700/targetrpc" -d '{"jsonrpc":"2.0", "id":0, "method":"create-filtered-volume", "params":{"name":"fvol1", "device":"/dev/sde", "filters":["xor"]}'
```

Since only the "method" and "params" field are significant, the descriptions of the methods which follow include only the parameters of the method, i.e., the possible contents of the "params" field. Similarly, the sample responses show only the "result" portion of the JSON object returned.

The following are the Konnector API primitives used by the SDS gateway to attach storage to a consumer node and build a filter stack terminated by a top of filter stack volume.

iSCSI initiator commands

1. get initiator name
2. discover portal
3. display discovery
4. display discovery summary
5. delete discovery
6. login target
7. logout all targets
8. display node summary
9. delete node
10. delete all nodes
11. display session
12. purge
13. create filtered volume
14. delete filtered volume

Filtered volume commands

1. create filtered volume
2. delete filtered volume

10.1.1 iSCSI initiator commands

get-initiator-name Get the iSCSI initiator IQN of the Konnector node. No parameters.

Example response: "iqn.1994-05.com.redhat:b0d684d83bb4"

discover-portal Discover all targets for a given iSCSI discovery portal.

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

auth-method null (default), "chap" or "mutual-chap"

username used only with auth-method "chap" or "mutual-chap"

password used only with auth-method "chap" or "mutual-chap"

username-in used only with auth-method "mutual-chap"

password-in used only with auth-method "mutual-chap"

Example response:

```
{
  "iqn.2004-04.com.mpstor:mb-vol-1": {
    "192.168.2.162": {
      "interface": "default",
      "portal": [
        "192.168.2.162",
        3260,
        1
      ]
    }
  },
  "iqn.2004-04.com.mpstor:mb-vol-2": {
    "192.168.2.162": {
      "interface": "default",
      "portal": [
        "192.168.2.162",
        3260,
        1
      ]
    }
  }
}
```

display-discovery Display all data for a given discovery record.

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

Example response:

```
{
  "sendtargets": {
    "address": "192.168.122.239",
    "auth": {
      "authmethod": "None",
      "password": "",
      "password-in": "",
      "username": "",
      "username-in": ""
    },
    "discoveryd-poll-inval": "30",
    "iscsi": {
      "MaxRecvDataSegmentLength": "32768"
    },
    "port": "3260",
    "reopen-max": "5",
  }
}
```

```
    "timeo": {
      "active-timeout": "30",
      "auth-timeout": "45"
    },
    "use-discoveryd": "No"
  },
  "startup": "manual",
  "type": "sendtargets"
}
```

display-discovery-summary No parameters.

Example response:

```
{
  "192.168.2.162": [
    3260,
    "sendtargets"
  ]
}
```

delete-discovery Delete discovery of targets at a given IP address.

input parameters and description

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

login-target Login to a given target.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

auth-method null (default), "chap" or "mutual-chap"

username used only with auth-method "chap" or "mutual-chap"

password used only with auth-method "chap" or "mutual-chap"

username-in used only with auth-method "mutual-chap"

password-in used only with auth-method "mutual-chap"

Example response:

```
"/dev/sdb"
```

logout-target Logout for a given target.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

logout-all-targets Logout for all targets.

No parameters.

display-node Display all data for a given node record.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

display-node-summary Display data for all node records.

No parameters.

delete-node Delete a given node record.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

delete-all-nodes Delete all node records.

No parameters.

display-session Display all data for a given session.

targetname | null (default) to display all sessions hostname | null (default) to display all sessions

purge Delete all records.

No parameters.

Filtered volume commands

create-filtered-volume Create a new volume by adding a stack of block storage filters to an existing block device. The filtered device is returned.

input parameters and description

name a unique name to assign to the underlying target created

device the device to which to add the filters e.g., "/dev/sda"; default null

serial alternative identification of device, e.g., "0xfed70573ae21"

filters a list of filters, e.g., ["xor"]; default null

handler need not be specified - default "mp-filter-stack" Example response:

"/dev/sdb"

delete-filtered-volume Delete a volume created using create-filtered-volume().

input parameters and description

name the unique name assigned in create-filtered-volume

10.2 Terminology

SAN A storage area network which is used to transfer data from storage arrays to consumer nodes usually servers running a storage centric or compute centric application. SANs are characterised by a protocol such as Ethernet, Fibre Channel, Infiniband, a speed 40G, 10G, 1G for Ethernet or 4G, 8G or 16G for Fibre Channel.

Software Defined Storage (SDS) A technology which virtualises storage hardware and allows users of that storage to provision storage capacity with defined properties without knowing anything about the underlying hardware infrastructure. SDS uses two three core concepts A) A logical to physical mapping of the storage hardware and storage SAN B) An out of band automation process that receives requests for storage and provisions this storage from storage arrays C) An in-band software layer that provides to the consumer a virtual block device (VBD) that is mapped to a physical storage array volume (SAV).

SDS Gateway An SDS Gateway is a software controller that interfaces to an upper level API such as CINDER OpenStack and a lower level storage controller such as a storage array or an SDS controller.

Storage Array A physical device that manages disk media (Solid State or spinning disks) and exports that storage over a SAN connection.

Storage Array Volume A volume on a storage array exported on a SAN. SAVs are usually built on a RAID made up of several disks.

Virtual Block Device A Virtual block device is a device attached to a storage application which is mapped to a storage array volume. A VBD may be mapped to a single SAV or multiple SAVs. This mapping is part of the SDS in-band software layer.

Storage Filters Storage filters are software functions inserted by the SDS controller between the SAV and the VBD on a compute node. Filters can implement a wide range of functions specific to the application using the VBD.

10.3 FIO test output

```
##### Sample FIO TEST REPORT #####
/dev/sdau: (g=0): rw=randread, bs=128K-128K/128K-128K/128K-128K, ioengine=libaio, iodepth=4
/dev/sdaw: (g=0): rw=randread, bs=128K-128K/128K-128K/128K-128K, ioengine=libaio, iodepth=4
fio-2.2.10
Starting 2 processes
/dev/sdau: (groupid=0, jobs=1): err= 0: pid=28225: Sun Nov  6 22:54:03 2016
  read : io=5185.0MB, bw=44243KB/s, iops=345, runt=120005msec
    slat (usec): min=8, max=255, avg=18.84, stdev= 8.71
    clat (msec): min=4, max=24, avg=11.55, stdev= 1.75
    lat (msec): min=4, max=24, avg=11.57, stdev= 1.75
    clat percentiles (usec):
      | 1.00th=[ 6816], 5.00th=[ 8640], 10.00th=[ 9280], 20.00th=[ 9792],
      | 30.00th=[10816], 40.00th=[11712], 50.00th=[11968], 60.00th=[12224],
      | 70.00th=[12480], 80.00th=[12864], 90.00th=[13376], 95.00th=[13888],
      | 99.00th=[15040], 99.50th=[15680], 99.90th=[18048], 99.95th=[20096],
      | 99.99th=[24192]
    bw (KB /s): min=40878, max=49976, per=50.27%, avg=44286.16, stdev=1668.83
    lat (msec) : 10=22.44%, 20=77.50%, 50=0.05%
  cpu          : usr=0.23%, sys=0.90%, ctx=18258, majf=0, minf=274
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
    submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    complete   : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    issued    : total=r=41480/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
    latency    : target=0, window=0, percentile=100.00%, depth=4
/dev/sdaw: (groupid=0, jobs=1): err= 0: pid=28226: Sun Nov  6 22:54:03 2016
  read : io=5139.0MB, bw=43850KB/s, iops=342, runt=120008msec
    slat (usec): min=8, max=192, avg=18.44, stdev= 8.24
    clat (msec): min=4, max=34, avg=11.65, stdev= 1.73
    lat (msec): min=4, max=34, avg=11.67, stdev= 1.73
    clat percentiles (usec):
      | 1.00th=[ 7008], 5.00th=[ 8768], 10.00th=[ 9536], 20.00th=[10048],
      | 30.00th=[11072], 40.00th=[11840], 50.00th=[12096], 60.00th=[12352],
      | 70.00th=[12608], 80.00th=[12736], 90.00th=[13248], 95.00th=[13760],
      | 99.00th=[15424], 99.50th=[15936], 99.90th=[20608], 99.95th=[22656],
      | 99.99th=[28032]
    bw (KB /s): min=38223, max=49053, per=49.83%, avg=43895.19, stdev=1583.64
    lat (msec) : 10=19.65%, 20=80.23%, 50=0.12%
  cpu          : usr=0.27%, sys=0.85%, ctx=17181, majf=0, minf=275
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
    submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    complete   : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    issued    : total=r=41112/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
    latency    : target=0, window=0, percentile=100.00%, depth=4

Run status group 0 (all jobs):
  READ: io=10324MB, aggrb=88092KB/s, minb=43849KB/s, maxb=44243KB/s, mint=120005msec, maxt=120008msec

Disk stats (read/write):
  sdau: ios=41452/0, merge=0/0, ticks=475828/0, in_queue=475852, util=99.48%
  sdaw: ios=41084/0, merge=0/0, ticks=476372/0, in_queue=476380, util=99.58%
#####
```

10.4 BitRev Filter example

Filter Source file

```
// Sample filter: bitrev. Reverses each byte in input buffer (in place) providing mirror image of source byte.
#include <stdio.h>
#include "bitrev_filter.h"

int g_flag = 0;
char *filter_name = "BITREV";

int pass_args( char* my_arg_list )
{
    int args_set = 0;
    return args_set;
}

char *get_name()
{
    return filter_name;
}

void print_bind_msg(char *msg)
{
    printf("%s Bit Reversal Filter\n", msg);
    return;
}

int get_flag()
{
    return g_flag;
}

void set_flag(int flag)
{
    g_flag = flag;
    return;
}

void write_xform( void* buf, unsigned long cnt, unsigned long offset, bool* doWrite )
{
    unsigned long i;
    unsigned char* my_buf = (unsigned char*)buf;
    for(i=0; i<cnt; i++)
    {
        my_buf[i] = (my_buf[i] & 0xF0) >> 4 | (my_buf[i] & 0x0F) << 4;
        my_buf[i] = (my_buf[i] & 0xCC) >> 2 | (my_buf[i] & 0x33) << 2;
        my_buf[i] = (my_buf[i] & 0xAA) >> 1 | (my_buf[i] & 0x55) << 1;
    }
    *doWrite = true;
}

void read_xform( void* buf, unsigned long cnt, unsigned long offset)
{
    unsigned long i;
    unsigned char* my_buf = (unsigned char*)buf;

    for(i=0; i<cnt; i++)
    {
        my_buf[i] = (my_buf[i] & 0xF0) >> 4 | (my_buf[i] & 0x0F) << 4;
        my_buf[i] = (my_buf[i] & 0xCC) >> 2 | (my_buf[i] & 0x33) << 2;
        my_buf[i] = (my_buf[i] & 0xAA) >> 1 | (my_buf[i] & 0x55) << 1;
    }
}

int pre_read(void* buf, unsigned long cnt, unsigned long int offset, unsigned int fdesc, bool * doRead)
{
    *doRead = true;
    return 0;
}
```

Filter Header file

```
/* flags.h "Header file" */
#include <stdbool.h>
```

```
void print_bind_msg(char *msg);
int get_flag();
void set_flag(int flag);
void write_xform(void* buf, unsigned long cnt, unsigned long offset, bool* doWrite);
void read_xform(void* buf, unsigned long cnt, unsigned long offset);
int pre_read(void* buf, unsigned long cnt, unsigned long int offset, unsigned int fdesc, bool * doRead);
int pass_args( char* raw_arg_list );
```

```
Filter make file
gcc -Wall -fPIC -c bitrev_filter.c
gcc -shared -Wl,-soname,lib_bitrev.so.1 -o bitrev.so bitrev_filter.o
```

Filter make file

```
gcc -Wall -fPIC -c bitrev_filter.c
gcc -shared -Wl,-soname,lib_bitrev.so.1 -o bitrev.so bitrev_filter.o
```

References

- [1] FORTH, IBM, BSC, INTEL, NEUROCOM, "IOLanes FP7 EU Project 248615." <http://www.iolanes.eu/>.
- [2] M. F. Oberhummer, "LZOP compressor." <https://www.lzop.org/>.
- [3] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3." RFC 1950 (Informational), May 1996.