Percipient StorAGe for Exascale Data Centric Computing

FETHPC1 - 671500

WP3 Services and tools

D3.1 HSM for SAGE: Concept and Architecture Report

SAGE_WP3_HSM_for_SAGE_Concept_and_Architecture_v1.0

Scheduled Delivery: 01.03.2016
Actual Delivery: 26.02.2016

Version 1.0

Responsible Partner: CEA

Revision history:

Date	Editor	Status	Version	Changes
27.01.2016	Thomas Leibovici	Draft	0.1	Initial Draft
03.02.2016	Thomas Leibovici	Draft	0.2	Completed version
04.02.2016	Thomas Leibovici	Draft	0.3	Ready for WP3 internal review
08.02.2016	Nathan Rutman	Draft	0.4	Rephrasing and clarifications.
08.02.2016	Thomas Leibovici	Draft	0.5	Inputs from Henri Doreau, J-C. Lafoucrière, Nathan Rutman.
09.02.2016	Thomas Leibovici	Draft	0.6	Inputs from Philippe Deniel.
18.02.2016	Thomas Leibovici	Final	1.0	Project review

Authors
Thomas Leibovici (CEA), Henri Doreau (CEA), J.-C. Lafoucrière (CEA), Philippe Deniel (CEA), Nathan Rutman (Seagate), Nikita Danilov (Seagate).

Internal Reviewers
Sai Narasimhamurthy (Seagate)

Acknowledgements
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671500

Glossary of Acronyms

Acronym	Definition
CASTOR	CERN Advanced STORage manager
D	Deliverable
DMF	Data Migration Facility (SGI software)
DoE	U.S. Department of Energy
DRS	Document Review Sheet
DTM	Distributed Transaction Manager (MERO component)
EC	European Commission
FDMI	File Data Manipulation Interface (MERO component)
FOL	File Operation Log (MERO component)
HPC	High Performance Computing
HPSS	High Performance Storage System (IBM software)
HSM	Hierarchical Storage Management
I/O	Inputs and Outputs of computation (data)
IT	Information Technology
MPP	Massively Parallel Processing
NBA	Non-Blocking Availability (MERO feature)
NVRAM	Non-Volatile Random Access Memory
RAM	Random Access Memory
SAGE	Percipient StorAGe for Exascale Data Centric Computing
SSD	Solid State Disks
SQL	Structured Query Language
TSM	Tivoli Storage Manager (IBM software)
PM	Project Manager
PO	Project Officer
WP	Work Package

Table of Contents

List of Figures

List of Tables

1. Executive Summary

As supercomputers become more powerful, they produce more data. To sustain the increasing data production, storage systems have to deliver higher throughput, and must be more capacitive to store the huge amount of generated data.

Hierarchical Storage Management (HSM) makes it possible to combine these objectives by mixing and taking advantage of various storage technologies, from the fastest to the more capacitive ones. It enables the design of high-performance and cost-effective storage architectures, which is a requirement for Exascale systems.

SAGE WP3.1 aims to design an extremely scalable and integrated HSM solution, capable of managing many storage tiers, handling billions of objects, and implementing smart and flexible data movement policies.

In this initial phase of the project, we identified the requirements of such a solution. We then defined its components, and their interactions with other components and actors. Next steps will be to complete the design in details, and implement a Proof of Concept to validate the designed solution.

2. Introduction

This document describes the concept of Hierarchical Storage Management, and how it can help designing performant and scalable storage systems for Exascale.
After studying strength and weaknesses of current HSM implementations, it will propose an innovative HSM solution, capable of managing billions of objects and Exabytes of data in the Exascale I/O regime.

In this section, we introduce the concept of Hierarchical Storage Management, and how it can be used to implement extreme performance and high capacity storage systems at a reasonable cost.

2.1. Benefits of HSM

2.1.1. Taking advantage of various storage technologies

In the history of computing and data storage, systems always had to handle multiple storage technologies: processor cache, memory, hard drive disks, magnetic tapes, etc. During the last decades, new storage technologies also emerged, like Solid State Disks (SSDs) and Non-Volatile Memory (NVRAM), filling the gap between "legacy" storage technologies, and enabling new usages. Each of these technologies has different characteristics in terms of speed, capacity, cost, and reliability. So storage architects now have a wide variety of technologies at their disposal to design storage systems.

The following table gives a rough estimate of the characteristics of the main storage technologies available today:

Technology	Latency	Bandwidth	Capacity/unit	Cost/Gbyte
RAM	~100ns	x10 GB/s	2-16 GB to 128GB	€€€€€
NVRAM	~10us	x1 GB/s	Up to x100 GB	€€€€
SSD	~150us	200-500 MB/s	Up to 1TB	€€€
Hard Drives	~10ms	50-300 MB/s	1 to 8 TB	€€
Tapes	Seconds	100-200 MB/s	2 to 8.5 TB	€

Table 1 Comparison of the main storage technologies today.

2.1.2. Caching and Hierarchical Storage Management

A significant part of theoretical research in computing is related to data *caching*. Most caching mechanisms rely on the same idea: a system is made of multiple storage device types with different characteristics in terms of bandwidth, latency, capacity, and cost. To take advantage of this variety of media, a classical approach is to use the fastest technology (which is often the most expensive) as a *cache* for the slower storage devices (cheaper and more capacitive). A common example of caching in operating systems is accelerating accesses to a hard drive by caching data in memory (RAM). In a cache, mechanisms are implemented to load data from the slow device to the cache, to synchronize data to it after a modification, and to release the cache when it is no longer used.

Hierarchical Storage Management generalizes the concept of data caching: the different storage technologies are organized hierarchically in *tiers*, from the fastest to the slowest. Data is generally accessed in the fastest tier, thus delivering the best access performance to user applications. Then the least recently or frequently used data is progressively migrated to lower tiers that offer larger storage capacities.

Consequently, Hierarchical Storage Management is a cost-effective solution to benefit from both the performance of expensive storage technologies, and the large capacity that can be afforded with cheaper storage.

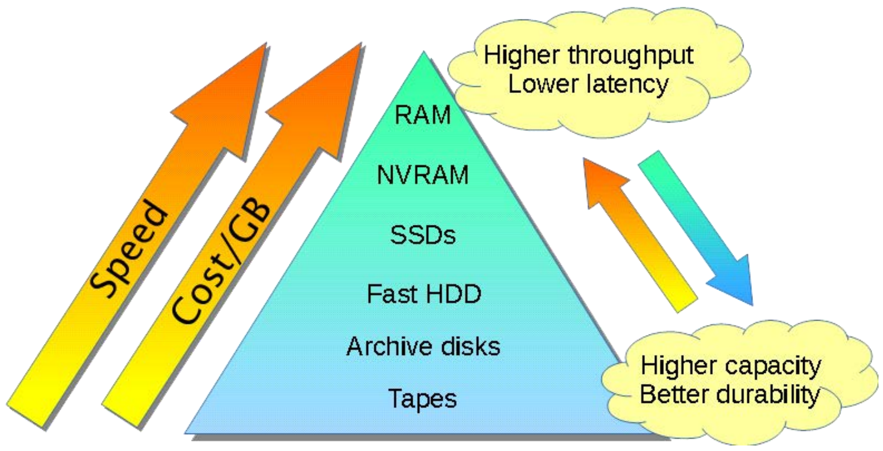

Figure 1: HSM combines the strengths of multiple storage technologies to implement performant and cost-effective storage.

2.2. State of the Art

Today, the two leading HSM solutions (HSMs) in largest HPC systems are DMF (1), developed by SGI, and HPSS (2), developed by a consortium of American's DoE labs and commercialized by IBM. These two HSMs have been used on major computing centers of the top500 (3) for decades. Other HSMs like SAM/QFS (Oracle) and TSM (IBM) are also widely used on mid-range or smaller computing centers.

2.2.1. 1990's: development of modern HSMs

DMF and HPSS were developed in the 90's. At that time, the revolution of massively parallel processing (MPP) resulted in a boom of data production. To handle the increasing volume of data, magnetic tape was the most economic technology, as hard drive disks were much more expensive. But tapes are not adapted to HPC I/O patterns. They are intrinsically sequential: only a single file can be read or written at once, in a sequential way. Disks are much more adapted to HPC workloads, as they enable parallel accesses, random I/O patterns and re-writes. So there was a big need for storage systems that could handle these two kinds of media transparently.

To address these needs, several HSMs were developed in those years:

- In 1988, **Tivoli Storage Manager** (initially released as *Workstation Data Save Facility, WDSF*) was implemented by IBM. Initially designed as a backup solution, it progressively evolved to an HSM system. Today, it is mostly used in the IT world, but is no longer used in large HPC systems.

- In 1989, SGI released the first version of its Data Migration Facility (**DMF**). DMF is a single-host HSM server that uses a local file system on disk as the first tier, and migrates old data to a second tier on tapes. The remote user interface is a NFS server that provides POSIX access to remote machines.

- In the mid-90's, DoE labs worked on a new model of mass storage to enable more parallelism. This model was proposed in 1993, and is entitled "Mass Storage System Reference Model" (4). This model was implemented as **HPSS** (High Performance Storage System) in 1995. HPSS has been the first parallel HSM. It enabled much more performance and scalability than others before, by providing parallel access to disks and tapes through multiple servers. HPSS provides a POSIX-like interface to its namespace through a NFS server which is notably inefficient for I/O operations. It provides better data access performances by a specific FTP implementation (pftp), or its native data transfer protocol (*mover protocol*).

  HPSS migration policies are based on a couple of parameters like age of files, and free disk space.

- The "Mass Storage System Reference Model" (4) was also implemented by CERN as an HSM named **CASTOR** (CERN Advanced STORage manager), in production since 2001.

2.2.2. 2000's: The Teraflop Era

At the beginning of the 2000's, the largest compute clusters reached a compute power of 1 Teraflops[1]. At that time, HSMs had a central role in computing centers: they provided a convenient place where users could store their huge amount of computation data in a central location, accessible from all machines (compute clusters, post-processing clusters, visualization facilities, etc).

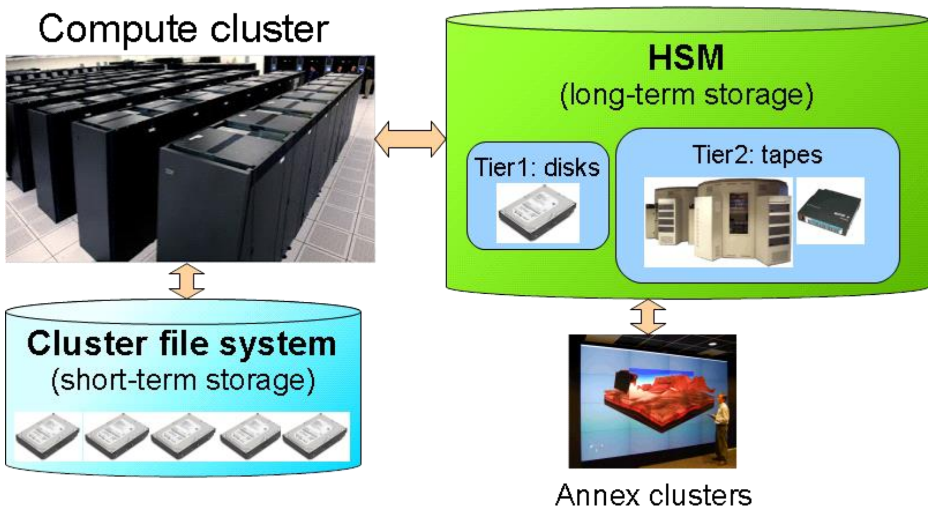

Figure 2: Role of HSMs in 2000's computing centers

2.2.3. 2010's: The Petaflop Era

As compute clusters reached 1 Petaflops[2], legacy HSMs showed their limits. They could hardly handle simultaneous accesses by thousands of compute nodes. It appeared that only parallel file systems could handle such levels of concurrency.

Parallel file systems were enhanced to support data migration to external storage backends, in particular to legacy HSMs:

- GPFS (parallel file system from IBM), implemented bindings to IBM's HSMs (TSM and HPSS).
- Lustre (5) (open-source parallel file system) implemented a backend-agnostic migration mechanism, thus making it possible to move data to any underlying HSM (DMF, HPSS, etc).

[1] 1 Teraflops = 10^{12} Floating-point Operations Per Second.
[2] 1 Petaflops = = 1000 Teraflops = 10^{15} Floating-point Operations Per Second.

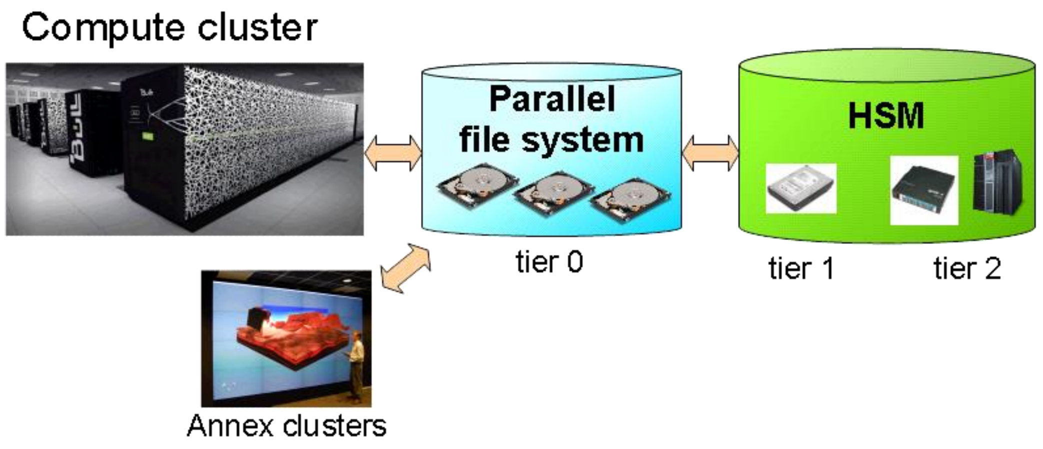

Figure 3: 2010's: Integration of HSMs as backends of Parallel file systems

This HSM integration in parallel file systems also brought innovations in scheduling data movements. These systems have *Policy Engines* that enable the definition of fine-grained policy rules using various criteria on entries.

- GPFS policy engine allows system administrators to define migration rules using an SQL-like language to match entry attributes. These rules are applied by scanning the GPFS metadata to select entries matching the specified rules.

- In the Lustre open-source ecosystem, *Robinhood* Policy Engine (6) allows system administrators to define policies based on various entry attributes like owner, path, name, size, age of last access or modification... This policy engine takes advantage of an innovative feature of Lustre to avoid scanning the filesystem namespace (which is a known scalability limitation for handling billions of entries): Lustre reports incremental changes in the filesystem to the policy engine through a *change log* mechanism.

2.2.4. 2020's: Dealing with Exaflops

By 2020, compute clusters are expected to reach Exaflop[3]. Like other scale changes before, it is already expected that previous generation of storage systems have to evolve to deal with the new I/O workload.

To help handling the increase of data throughput, new storage technologies have emerged, like SSDs and NVRAM. Integrating those high-performance technologies to

[3] 1 Exaflops = 1,000 Petaflops = 10^{18} Floating-point Operations Per Second.

future storage system will be mandatory to sustain the data production of these future clusters.

Some solutions are currently under development to adapt today's systems to this new workload. Again, these solutions are about adding new (yet another) software layers on top of older storage technologies. This is the case of DAOS (7) which aims to implement support of NVRAM burst-buffers on top of Lustre, like represented on Figure 4.

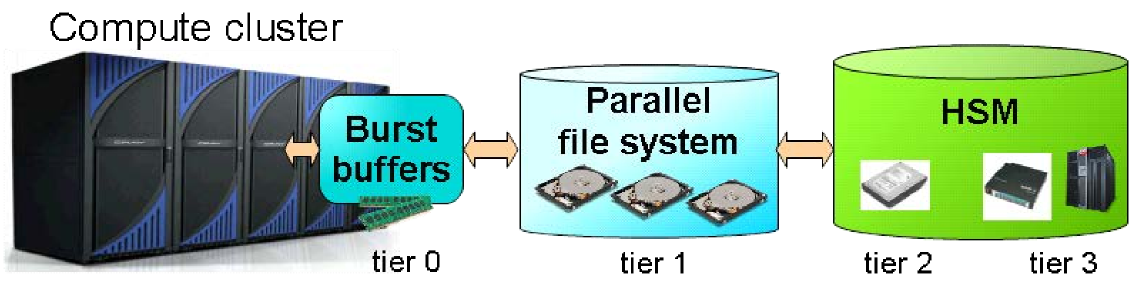

Figure 4: Overview of other Exascale storage projects

2.3. HSM for SAGE: objectives

While other projects continue stacking old and newer technologies to work around limitations of old technologies, SAGE proposes a new integrated architecture, adapted to Exascale from end-to-end.

It aims to implement an integrated solution, capable of managing many levels of storage, from very high performance storage (memory, NVRAM, etc.) to archival storage, with an optimal use of today's storage technologies.

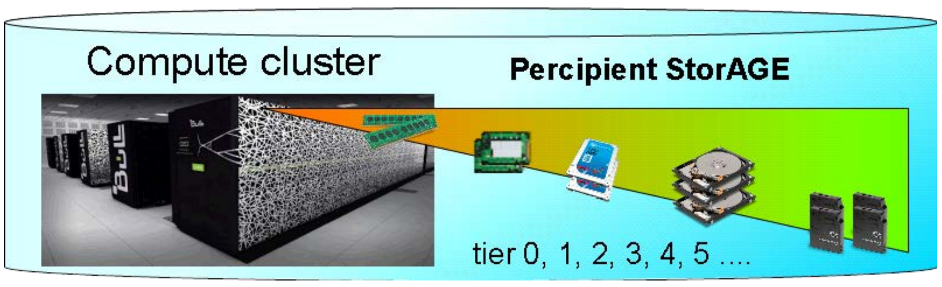

Figure 5: SAGE integrated HSM model

There is also room for major improvements in the domain of HSM policies. Today's HSM policy engines consider each filesystem entry independently, without any global understanding of applications and computing center environment. As part of the HSM implementation for SAGE, a new generation of policy engine will be defined. It will be smarter, by implementing machine-learning algorithms to learn from past application access, so it can then take optimal HSM decisions. It will also be aware of the computing center environment, by interacting with components like job schedulers.

A SAGE objective is also to bring the data close to compute codes, by managing storage that could potentially even be in the compute nodes (like NVRAM), or by shipping the computation functions up to the storage servers, thus dramatically reducing the cost of network communications for accessing the data.

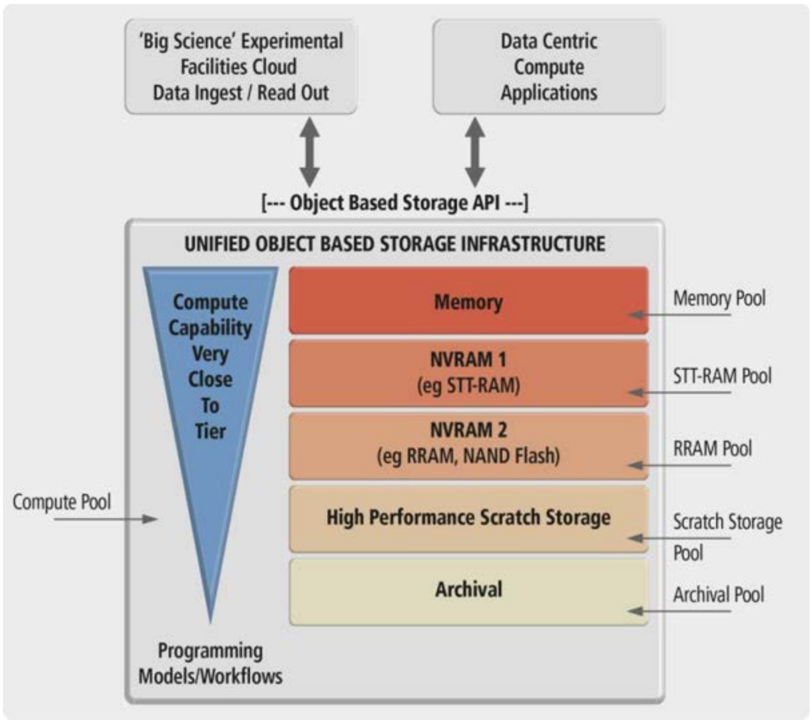
'Big Science' Experimental Facilities Cloud Data Ingest / Read Out
Data Centric Compute Applications
[--- Object Based Storage API ---]
UNIFIED OBJECT BASED STORAGE INFRASTRUCTURE
Compute Capability Very Close To Tier
Memory
NVRAM 1 (eg STT-RAM)
NVRAM 2 (eg RRAM, NAND Flash)
High Performance Scratch Storage
Archival
Memory Pool
STT-RAM Pool
RRAM Pool
Scratch Storage Pool
Archival Pool
Compute Pool
Programming Models/Workflows

Figure 6: SAGE architecture

The proposed solution will drop the scalability limitations of POSIX (8) by adopting a new semantics of object-based storage, while still maintaining backwards POSIX compatibility[4] for legacy purposes.

[4] This is the purpose of SAGE WP3, Task 3.2 that aims to provide an access to SAGE through a standard and parallel protocol: pNFS (9).

3. Architecture

This section deals with design aspects and describes the architecture of an HSM solution for SAGE.

3.1. Data Flows in Deep Storage Hierarchies

Hierarchical Storage Management involves multiple storage technologies, usually organized in a particular order, from the fastest and least capacitive to the slowest and most capacitive.

This section defines the possible data movements in such a hierarchy, and the requirements for implementing them.

3.1.1. Placement of Incoming Data

In most HSMs, data is always written initially to the fastest tier. This basic approach can be inefficient in some cases.

For example, if the written data exceeds the capacity of the tier, data must immediately be moved to the lower tier as soon as it arrives on the fast tier. In case of a sequential write operation, there is no interest in doing this:

- The sustained write throughput will be limited by the lower tier speed.
- This results in consuming all the space in the fast device, and so prevents other data accesses to benefit from it.
- It causes bandwidth waste; data is first written to the fast tier, thus consuming write bandwidth on it. Then it must be read to be copied to the lower tier, thus consuming read bandwidth on it. And finally it must be written to the lower tier, which consumes write bandwidth on this lower tier. Directly writing the data to the lower tier would avoid the first two operations (write then read in the fast tier).

In such a case, it would be more efficient to directly write the data in a lower tier.

For instance, it may not be optimal to perform a sequential write of 500GB to a 128GB NVRAM device.

Generally, if the characteristics of an I/O operation match device characteristics of a low tier, it may be preferable to send the data directly to this tier, to leave fast resources available for data accesses that really require them.

Taking such a placement decision requires some knowledge about the current I/O operation. In classical systems, there is no way for applications to indicate an estimated data volume and access pattern at file creation time.

To enable such optimization in SAGE, hints about I/O accesses are required:

- Hints can be passed by the application to the storage system API.
- Some hints can be collected in client environment (e.g. executable name, environment variables set by the job scheduler).

- Storage system can learn I/O patterns of applications, based on past application runs.

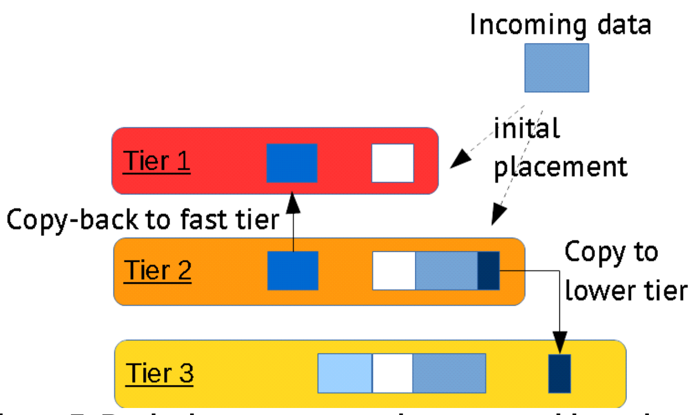

Figure 7: Basic data movements in a storage hierarchy

3.1.2. Migrating data to slower tiers

As fast tiers continuously fill up with new incoming data, unused or less frequently used data must be moved to slower tiers to make room for newly accessed data.

The main concern about this operation is to select the right data to be moved, without penalizing future accesses, and to avoid resource waste by copying a frequently modified object multiple times:

- If frequently accessed data is moved to a slower device, next accesses to this data will be less performant. So taking a bad decision of data movement can significantly impact application performance.
- If a data is modified while (or after) it is copied to a lower tier, the target copy needs to be resynchronized before removing the source data from the up-tier. This can result in a bandwidth waste by doing multiple copies of some data blocks. Or in the worst case, the copy process may not converge.

Again, the application is in a good position to know if it will modify any data in the future, so the copy decision should be influenced by the application. However this is not enough because:

- Copy to lower tier is possibly (preferably) run for “cold”[5] data: it can often apply to data of past application runs.
- This does not allow HSM to coordinate copy streams from all applications. Copy streams from all applications are in competition for data transfer bandwidth, so an overall fair-share mechanism is required.

The copy workflow can be huge as most of the produced data is eventually copied to a lower tier. So it seems important to have an overall control on this stream, to limit its

[5] Concept of data temperature: the more a data is accessed, the hotter it is considered.

impact on application performance and on the global system load. This leads to preference for a batched copy workflow managed by the system:

- To bulk copy a large amount of data efficiently.
- To control the global load on the system.
- To take copy decisions based on all present data in the tier, not only the data from one application.
- As the copy decision is asynchronous, it can implement more complex algorithms without inducing latency on application I/Os.
- The copy decision may rely on machine learning algorithms to predict how long data should remain in the current tier, and determine the next tier it should be copied to.

Such a batched copy mechanism can still take application hints into account: if an application declares it will not modify an object anymore, the system can integrate this object into its batches of copies, possibly with a higher priority.

3.1.3. Recalling data to a fast tier

An application may desire to read or write data which is located in any tier, not necessarily the fastest one.

If the access is rare, the operation can be performed immediately without staging data to a faster tier, thus saving useless copies, and avoiding bandwidth waste.

If the application plans to access data intensively (e.g. heavy random I/O on a whole dataset), or if the system detects an heavy I/O pattern on a data located in a low tier, it can be worthwhile copying the data to a faster tier, to offer the application a better throughput and a lower access latency.

When an application explicitly requests an intensive I/O access, reading back data from a lower tier is latency critical as it happens on “hot” data that needs to be accessed as fast as possible. So the HSM implementation should avoid blocking operations and introducing latencies in this case.

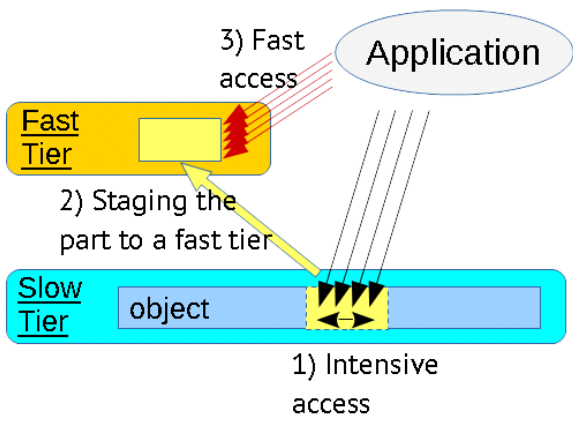

Figure 8 Copy- back of intensively

used data to a fast tier

Several options can be considered:

- **Synchronous recall:** Copy-back is triggered when an application requests a faster data access. There are 2 possible behaviours in this case:
  - **Blocking recall:** Access is blocked until the requested data is available on the fast tier.
  - **Non-blocking recall:** Even if the data is not yet copied back to the requested fast tier, data should be read from a lower tier where it is available and immediately returned to the application.

  Choosing one strategy or the other greatly depends on the storage media characteristics: if the low tier consists of SSDs, the non-blocking strategy should be more efficient. But if the low tier consists of tapes, it will not support concurrent accesses (simultaneous copy-back and application accesses) so the blocking case is more appropriate.
- **Asynchronous recall:** Data access is done wherever the data is located. The system analyses data accesses asynchronously and can decide to copy-back data which is intensively used.

3.1.4. Data management granularity

In previous sections, we have considered “data” movements in general. A related question is what granularity of data the HSM solution must handle: it can be single objects, parts of objects, or group of objects.

This section studies the pros and cons for each level of management granularity.

Groups of objects:

- Pro: It moves together objects that are used at the same time, resulting in more efficient bulk data movement, and pre-staging objects likely to be used.
- Con: Even if several objects are used together, they may not be accessed with the same IO patterns (e.g. metadata vs. unstructured data vs. logs…).

Individual objects:

- Pros:
  - Simpler implementation: HSM status can be managed as a few state bits.
  - If an application requests to read a part of a file, it is likely interested in other parts of it, so loading the whole object is a natural prefetching strategy.
  - It is adapted to media where the highest cost is to read the first byte, like tapes.
- Cons:
  - It causes a read-back of a whole object even for accessing a small part of it.
  - Huge objects may not fit in the highest tier as a whole (e.g. a 1TB database may not fit in a NVRAM device of a hundred GigaBytes). This leads to an over-consumption of space in the top tier.

  - This is not compliant with common UNIX utilities like 'file' – that only reads a small part of files, or with data formats that embed metadata or thumbnails (e.g. JPEG).
  - It causes re-write of whole objects to lower tiers, even if only a small part of them has changed.

Part of objects:

- Pros:
  - Efficient for managing partial read of objects, and partial modifications. No bandwidth waste.
  - Adapt to any size of device.
- Cons:
  - More complex: requires managing maps of object parts spread across multiple tiers.
  - Pre-fetching: application must give hints to preload a whole object, or the system must guess future application needs (read-ahead strategy).

On one hand, if the HSM solution can deal with arbitrary-sized parts of objects, it can also handle a part which is actually the whole object. On the other hand, managing groups of objects can be implemented as part of the smart scheduling of data movements.

The sizing of the manipulated parts should be driven by:

- Application accesses: If an application heavily accesses a particular part of an object (all operations are done in a given range of it), the part to be moved to a faster tier is inferred from application access pattern (possibly expanded for disk I/O alignment, and to minimize the size of layout metadata structures).
- Media characteristics: A given tier may specify a "preferred" data granularity. For example, it could prescribe a minimal size under which the device throughput is not optimal (e.g. a few Megabytes for disks, several Gigabytes for tapes, etc). The HSM solution should be aware of these storage characteristics, and choose the data movement size accordingly.

3.1.5. Interest of machine learning

Today's policy engines allow the system administrator to define data migration rules using various criteria.

Here is an example of simple HSM policy rule that an administrator could define:

➢ *"If an object has not been accessed in the last 30 days, move it to slower tier."*

This rule is very basic and do not take the various access patterns of applications into account. For example, some data may not be reused after 1 day, so it will waste space in the fast tier during the next 29 days. Also, if a large data is read every 2 months, it will be read from tape every time. As a result, it appears such a basic policy does not optimize the data flow.

To illustrate the complexity of optimizing data flows in a storage hierarchy, let's take an example of common HPC access pattern:

- When the code starts, it first creates an object named "*something*/run.log" which is a log file for debugging purpose.
- This application produces checkpoint files at regular interval called "*something*/checkpoint_<n>.out". These files are very large.
- It also generates result data at regular interval, in files called "*something*/result_<n>.out".

In this example, let's consider there are several user communities:

- Some are scientists: They run simulations for their researches. They never read the contents of the log files. But they intensively read the contents of result files during a period of about one week, and then keep them for years, and sometimes read them again.
- Some are code developers: They implement and test simulation codes. They read the contents of log files very often for debugging. They sometimes take a look at result data (to check if it is right), but rarely need it for more that 1 or 2 days. After that, they remove all data generated by their test.

If a system administrator studies the particular pattern of this application, and how it is used by the different communities, he can optimize the HSM policies for this application by defining smarter rules like:

- ➢ *"If a file is named 'run.log', and is generated by a user in 'developer' group, keep it on fast disk".*
- ➢ *"If a file is named 'run.log', and is generated by a user in group 'scientist', move it to archive storage."*
- ➢ *"If a file is named 'result*', and is generated by a user in group 'scientist', place it on fastest storage during 1 week, and archive it."*

  etc

To define a complete set of policies, system administrators needs do the same work for every application that runs on the system. They must also regularly refresh the list of applications running on the system, to take new patterns and new usages into account.

As this example shows, there is still much room for automation in the domain of policy engines! Machine-learning algorithms could help automate the whole process of pattern analysis, by correlating various criteria like applications, users, file types, and usages.

HSM components can collect useful information about applications:

- They can gather information from the user environment when it is invoked (e.g. for pool selection): application name and run arguments, user, groups, information from the scheduler etc.
- They can collect access information from the storage system, to characterize the I/O pattern, and lifetime of data.

Then, a machine learning algorithm can correlate those data and determine policies to optimize the data flow.

3.2. Core Storage Components

The core of Percipient StorAGE relies on “MERO” storage framework, which will be described in WP2 deliverables. Several components of this system have been identified as requirements to implement Hierarchical Storage Management.

3.2.1. Pools

In MERO, a *pool* is a collection of hardware and software resources (servers, processes, attached storage devices and network links), which together display certain prescribed availability and fault-tolerance characteristics.

In a multi-tiered architecture, pools can be used to partition a storage system into multiple levels, each pool consisting of a particular media type with homogeneous performance characteristics. So HSM tiers can be implemented by pools.

In the present HSM design, we only consider non-volatile storage. RAM (memory) is not considered as a storage pool managed by HSM: memory caching is managed independently by a specific mechanism of MERO.

Here is an example of pool configuration to implement hierarchical storage:

- Tier 0: NVRAM pool
- Tier 1: SSD pool
- Tier 2: disk pool
- Tier 3: archive pool

In current MERO design, applications can specify availability and fault tolerance requirements when they create transactions.
Additionally, for HSM, these requirements should be augmented by media-specific criteria (like latency and throughput) to let the system determine the optimal pool to access/store the data.
The pool concept could also be extended to add support for extra tiers that would be offline storage out of MERO, like a tape storage system. Such a storage system could be treated as a pool on condition that this external system provides information about fault-tolerance, thus fitting into the MERO pool management interface.

3.2.2. Layouts

Layouts describe how data is stored in the system. A layout consists of a group of extents of an object in a specific pool, having a specific data distribution strategy. For example, a distribution strategy can be “RAID1” - in this case, the data is mirrored on two devices. Another possible distribution is “parity de-clustered network RAID with 8+2 striping”.
Layouts prescribe how data should be placed in the system eventually, i.e. they do not describe transient caching mechanisms.

3.2.3. Composite Layouts

Mero’s composite layouts make it possible to manage multiple layout descriptions for multiple, potentially overlapping, parts of an object (called *extents*).

In a multi-tiered architecture, composite layouts can be used to keep track of multiple fragments of objects residing in the different tiers, and maintain information about their current status.

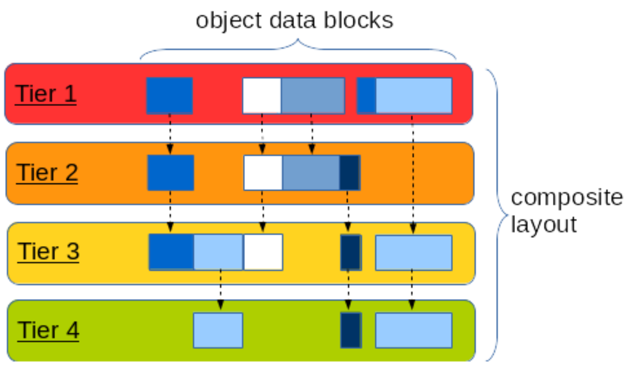

Figure 9: Using Composite Layouts for HSM

Figure 9 represents how composite layouts can be used to map fragments of data located in the different HSM tiers.

Composite layouts are used in a particular order; when an application wants to access a given block of data, it uses the first level of the composite layout where the block is available. This particularly matches the HSM need; by ordering composite layouts from the fastest tier to the slowest, applications will access in priority the fastest tier where the data is present.

Composite layouts can be modified in the same distributed transactions that encapsulate data modifications, thus ensuring consistency between data and the related mapping represented in composite layouts.

3.2.4. Non-Blocking Availability

This feature makes it possible to freeze a level of a composite layout, and to redirect all further write operations to a new layout level which can be either located in the same tier or a different tier.
This mechanism was originally designed to deal with server failures, by allowing a client to continue to write to an object even if one of the servers storing the object failed.
It appears this mechanism can also be used to manage hierarchical storage: it makes it possible to "freeze" (and continue to read) the data which is located in a low tier, and redirect write operations to a new layout which is located in a faster tier. More details about using this mechanism will be given in section 3.3.1 "Copy Mechanism".

3.2.5. Resource Manager

Resource Manager is a generic component of MERO. It can be used to control access to any component that can be considered as a resource: files, locks, quotas, disk space, bandwidth, etc.

The interface between the Resource Manager and other components is quite simple: a client queries a Resource Owner for a “credit” which is granted once the requested resource is available.

Resources managers can be involved in several aspects of Hierarchical Storage Management:

- Resource Managers keep track of free space on devices. They should report device usage to the HSM policy engine, so it can take appropriate policy decisions to move data to another tier, for example to release space on a full device.
- To ensure the consistency of composite layouts, their modification should be protected by a locking mechanism, which is implemented by a Resource Manager.

3.2.6. FOL and FDMI

FOL (File Operation Log) keeps track of all operations in MERO. For scalability reasons, this log is distributed across all MERO servers. This log contains crucial information for HSM, concerning object creation, modification, accesses, and I/O patterns.

FDMI (File Data Manipulation Interface) allows a user process running on any MERO machine to subscribe to particular FOL records matching a given filter. Such a process can then handle batches of records asynchronously without introducing latency to I/O operations. Records are stored persistently on the servers until the consumer acknowledges them, which ensures no record is missed by the subscriber.

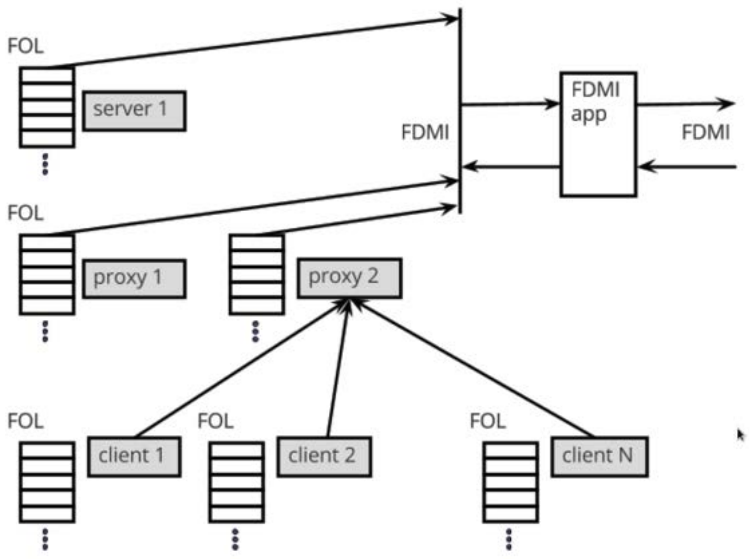

Figure 10: FDMI's MapReduce-style algorithm

For HSM, FDMI can be used:

- To collect data access information that can be necessary to make movement decisions.
- To determine if a file is modified while it is copied from one tier to another.
- To be kept aware of resource usage in the system.

So far, FDMI only reports FOL records for update operations. It must be extended to report read operations on objects, in order to keep HSM aware of read accesses. However, reporting all read accesses can impact the performance of the system, as read operations of applications would cause write operation to the File Operation Log of MERO. So we may consider an intermediary solution where FDMI would only report a summary of read accesses, or only report them periodically.

Information from Resource Managers could also be useful for an FDMI plugin to make HSM decisions: range locking information (including read operations) would help illuminate access patterns, and storage resources utilization could trigger cleaning of full tiers. It is therefore desirable to augment the FDMI implementation with Resource Managers events.

3.2.7. Distributed Transaction Manager (DTM)

Distributed Transaction Manager makes it possible to manage a set of operations distributed across multiple servers and storage resources. It ensures the atomicity of the group of operations in case of failure.

The DTM will be used to synchronize the data copy operation between tiers and the modification of composite layouts to reflect the new data location.

3.3. HSM Components

The section describes the specific new components that need to be implemented as part of SAGE WP3, Task 3.1.

3.3.1. Copy Mechanism

Moving data between tiers is a key feature of Hierarchical Storage Management.

Although moving data from a source object to another is a basic operation that can be done through a standard client API (i.e. Clovis API), the difficult aspects for HSM to implement are:

- Making the data movement transparent to applications and users, even if they are currently accessing the data.
- Ensuring object consistency and change durability while the copy is taking place:
    - No object modification must be lost while an object or extent is moved from one tier to another.
    - Read must be directed to the right data. In particular, it must reflect previous writes.
- Limiting the impact on access performance, by reducing the need and the duration of critical sections that must be protected by a lock.

The involved mechanisms to address these requirements have been identified during this preliminary phase of the project. They will rely on *Composite layouts* and *Non-Blocking Availability* (NBA) feature, previously described in section 3.2.

Copy-down

When an object extent is to be moved to a lower tier, its current layout is frozen (becomes read-only), just like NBA feature does, and new writes are redirected to a new layout in the same tier. The frozen version is used as the source of the copy, and the new layout is used as the preferred copy for application accesses.

This presents several advantages:

- The copy is done on a "snapshot" of the extent at a given point in time. Its contents do not change during the copy.
- Data remains accessible to the application. Read operations are directed to the right layout, depending if the related block is present in the new layout or the frozen one.
- Write operations that occur in the new layout are tracked by the layout structure itself. So if new writes occur during the copy, they can be later applied to the copy target, an incremental way.

The copy target is a layout located in the selected target pool. This layout can be integrated to the composite layout of the object once the copy completes.

The following table illustrates the state of the composite layout for a given extent when the copy-down operation completes:

Table 2: Example of extent layout when copy-down operation completes

Layout index	Location	Description	Details
0	Source pool	"Active" layout	Where application writes are directed
1	Source pool	Copy source	Frozen (read-only)
2	Target pool	Copy target	Inserted when the copy completes

Once the copy is completed:

- If the goal is only to pre-migrate an object without releasing space from source pool, we can leave the layout as-is. Layout 0 and 1 remains in the composite layout to keep track of changes compared to the layout copied to the lower tier.
- If the HSM decision is to move the object down-tier and drop the object from the fast tier, we can remove the source layout (index 1 in the example above).
  Level 0 can also be dropped if it contains no data (if no write occurred). Else, we can either:
  - Restart the operation with the subset of extents that have been written to level 0 (freeze them, create a new layout to collect write operations, and so on..)
  - Leave the layout as is, and integrate the newly written extents to the standard HSM decision workflow of the Policy Engine (see section 3.3.3)..

Copy-up (or copy-back)

When an object extent is to be staged to a faster tier (or if we want to prevent applications from accessing data in a very slow tier), the source copy is frozen - the same way we described previously, and a new layout is created in the (fast) target tier.

- Write operations from the application are directed to the fast tier.
- Application reads are preferably redirected to the fast tier (if the requested data is present), or else they are directed to the slow tier (which is read-only at this point).

At the same time, another layout is created in the fast tier. This layout is used as the copy target for the staging operation (no concurrence between the copy and application accesses).

The following table illustrates the state of the composite layout for a given extent when the copy-up operation completes:

Table 3: Example of extent layout when copy-up operation completes

Layout index	Location	Description	Details
0	Target pool	“Active” layout	Where application writes are directed
1	Target pool	Copy target	Inserted when the copy completes
2	Source pool	Copy source	Frozen (read-only)

When the copy completes (or at regular intervals), this target layout is integrated to the composite layout of the object with a lower priority than the layout which is currently used for handling application writes. This way, any write that occurred during the copy have priority on the data retrieved from the low tier.

Releasing an extent from a tier

An extent can be released from a tier if its data has been copied down-tier and if the active layout (the one where writes are redirected) is empty. This condition also applies to an extent that has been staged from a lower tier.

3.3.2. Pool Selection Service

The purpose of the Pool Selection Service is to determine the most appropriate pool where an object (or part of object) should be located, to deliver the best performance to applications, and ensure a fair sharing of resources between applications.

The most efficient location to access data greatly depends on the application access patterns; volume, required throughput and latency, random or sequential I/O, whether data is to be accessed frequently, or if it is written with no intent to read it in the short-term, etc.

Applications have the best knowledge of these characteristics; they know the purpose of each datum they handle, how critical it is to the application, what data may be reused, and so on. Applications should therefore be able to give hints to make it possible to select the most appropriate storage technology.

These hints would consist of functional requirements including:

- Whether the data is to be read or written
- Whether it will be re-read or re-written multiple times
- Throughput requirements
- Latency requirements
- Volume of data to be handled
- I/O pattern (sequential or random)
- Data access time (how long the data will be accessed)
- Data life time (how long data will reside in the system)
- Availability (data replication level and fault tolerance)

From this set of requirements, the Pool Selection Service must be able to determine the appropriate data placement.

But even if applications have a general knowledge of these parameters, it may be difficult for an application to give accurate specific values for all of them. Instead, the system administrator could define families of I/O patterns (e.g. “temporary data”, “checkpoint data”, “logs”, “result datasets”, etc) each of these families being associated with a range of values for these parameters. In this case, an application would just need to specify the family of I/O pattern for accessing data.

This system can be improved by integrating a machine-learning algorithm that would be able to predict the I/O pattern of an application by learning from past executions. The knowledge-base for this machine-learning can be fed by the pool service itself, and the HSM policy engine (see next section).

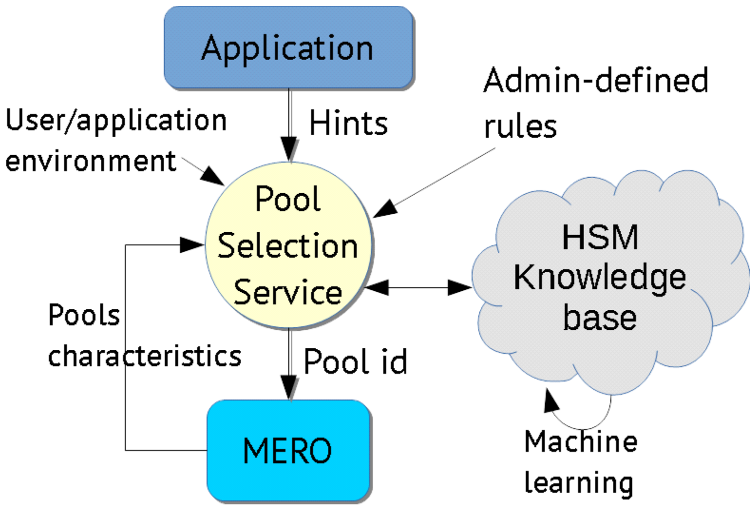

Figure 11: Overview of the Pool Selection Service

Nonetheless, data placement is latency critical as it is done synchronously in the application I/O workflow. So we cannot run expensive computations like machine-learning algorithms at this time. This should be done as an asynchronous background task, and should generate a ready-to-use placement function that can be used at no cost in the application workflow.

3.3.3. Policy Engine

The purpose of the HSM policy engine is to make movement decisions according to various criteria.

As output, it must determine:

- What object, part of object, or group of objects is to be moved from one tier to another.
- Where the data should be moved to (target tier).
- When (or how often) the data should be moved.
- Whether a copy of the data should remain in the source tier, or if it should be dropped after the copy.

As input, it can collect information from multiple sources:

- FDMI that reports all operations in the storage system.
- Resource and device usage information.
- Application hints (collected by Pool Selection Service, or passed to the policy engine through another mechanism).
- It can query an external component to determine the status of a given object (e.g. query the job scheduler about a given JOB_ID, to determine if the job that produced an object is still running).
- It can receive orders or events from external components.

Decisions are made following some guidelines:

- System administrator can define rules to trigger, prioritize, or forbid some data movements, and specify the direction and target of those movements.
- Policy Engine can implement machine-learning algorithms to predict application needs and take appropriate movement decisions.

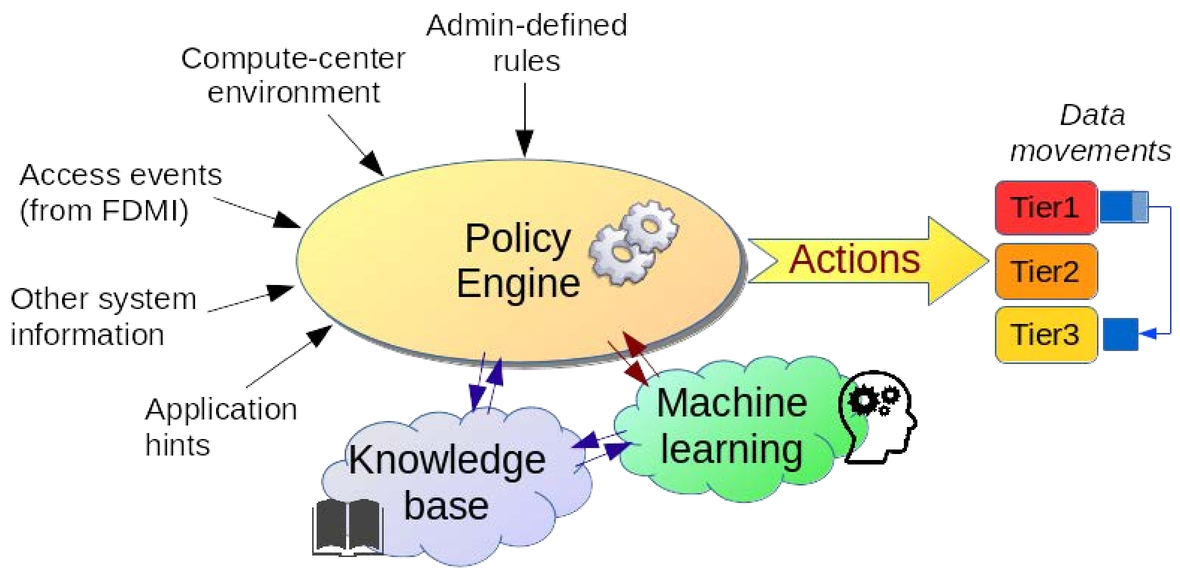

Figure 12: Big picture of the HSM Policy Engine

4. Relation to other Work Packages and Tasks

4.1. WP1: Application and Use

As stated in the present report, there will be interactions between applications and the HSM solution.

4.1.1. Interface Specification

When they access an object, applications will have to provide "hints" to help the system placing the data to the optimal storage tier. The following aspects will have to be specified in agreement with applications:

- Contents of HSM hints, and the ability of applications to provide them.
- Interface to specify HSM hints.
- Life cycle of data.
- Transparency of the HSM solution.

4.1.2. Benchmarking and Evaluation

WP1 defined a set of representative HPC applications. These applications implement different use-cases, and potentially have different data access patterns and workflows.

In later phases of the project, these representative applications can be used:

- To benchmark the HSM solution, and evaluate its ability to sustain Exascale I/O workloads.
- To evaluate its capacity to characterise data access patterns and take the appropriate data movement decisions.
- To evaluate the benefits of HSM data placement on application performance.

4.2. WP2: Percipient Storage

HSM solution will rely on some components and features of MERO, which is the core storage system of SAGE.

HSM will interact with WP2 by providing a list of requirements including:

- Required features and enhancements in the storage system to implement HSM (e.g. FDMI, pool selection, composite layouts, etc).
- Needed APIs to handle storage elements.

4.3. WP3: Services and Tools

HSM can interact with other tasks of WP3, in particular with the pNFS server (WP3 Task3.2):

- As the pNFS server will access the storage like any other application, it should implement the same interactions with HSM, by giving information about its access patterns, in order to locate the accessed object in the optimal tier.
- As the pNFS is a gateway to users, it is desirable that it implements a mechanism to pass application hints to HSM, through the pNFS protocol (e.g. by setting an extended attribute).
- The more contextual information is passed to the HSM policy engine, the smarter its movement decisions can be. So we should study what contextual information the pNFS server can pass to the HSM policy engine.

4.4. WP4: Programming Models and Analytics

4.4.1. Interface Specification

WP4 is involved in adapting MPI-IO middleware to the SAGE platform. As for WP1, specifying application interface to HSM should be done in agreement with WP4.
In particular, the IO middleware should be able to pass meaningful information to HSM services, to optimize data placement and IO performance.

4.4.2. I/O pattern analysis

Running an MPI-IO-based application on the SAGE platform will be a relevant benchmark of HSM, to evaluate its capacity to characterise data access patterns and take the appropriate data movement decisions.

4.4.3. Analytics & Machine learning

Task 3.1 and WP4 have a common interest in using machine-learning algorithms.
Thus, Task 3.1 could rely on WP4 achievements in machine-learning, and use it for its own needs of workload analysis and movement scheduling.

4.5. WP5: Integration and Demonstration

A test and validation platform has been designed as part of SAGE WP5, Task 5.1. This system consists of 4 storage tiers with the newest technologies: NVRAM, SSDs, fast disk and archive disks. As this hardware makes it possible to store data on various storage technologies with different characteristics of performance and capacity, it is a perfectly suited to test and validate Hierarchical Storage Management.

It will be used to validate different scenarios of hierarchical storage management, and to evaluate its ability to take the right movement decisions depending on the application workflow.

5. Conclusion and Next Steps

In this initial phase of the project, we focussed on defining the architecture and components of the HSM solution for SAGE. We identified the interactions with other components of SAGE, as well as functional requirements to implement hierarchical storage management.

The next step will be to draw up the detailed design of each of these components, by specifying particularly:

- Their internal structure.
- The involved algorithms.
- Their precise interface to other components and actors.
- How they achieve extreme scalability.
- How they provide high-availability.
- How they are integrated to the system.

Proof of Concepts will be implemented to validate these various aspects.

The results of these steps will be presented in deliverable D3.5 (M18).

6. References

1. Corp., Silicon Graphics International. SGI's Data Migration Facility. https://www.sgi.com/products/storage/idm/dmf.html.
2. HPSS - High Performance Storage System. http://www.hpss-collaboration.org/.
3. *Top 500 Supercomputer Sites.* [En ligne] www.top500.org.
4. Mass Storage System Reference Model. IEEE, 1993. http://www.computer-history.info/Page4.dir/pages/Storage.dir/storage.reference.model.pdf.
5. *Lustre official home.* http://lustre.org.
6. *Robinhood Policy Engine: project web site.* http://robinhood.sf.net.
7. Gorda, Bent. DAOS: an Architecture for Exascale Storage. 2015. storageconference.us/2015/Presentations/Gorda.pdf.
8. IEEE Std 1003.1. *Standard for Information Technology - Portable Operating System Interface.* www.unix.org/version3/ieee_std.html.
9. Network File System (NFS) Version 4.1 (pNFS). *RFC 5661.* January 2010. https://tools.ietf.org/html/rfc5661.