Storage Management in Smart Data Lake

Haoqiong Bian Bikash Chandra Ioannis Mytilinis Anastasia Ailamaki
EPFL
firstname.lastname@epfl.ch

ABSTRACT

Data lakes are complex ecosystems where heterogeneity prevails. Raw data of diverse formats are stored and processed, while long and expensive ETL processes are avoided. Apart from data-heterogeneity, data lakes also entail hardware-heterogeneity. Typical installations involve distributed infrastructures, where each node is possibly equipped with hardware of different characteristics. Especially for the case of storage, the various devices a node possesses can be organized in a hierarchy that defines a spectrum of performance-capacity-cost configurations. Given the various configurations and the volatile workload landscape, taking optimal placement decisions is a cumbersome task.

In this work, we propose a storage management solution for the *Smart Data Lake* [12] platform. The proposed system takes advantage of the available storage devices, while it abstracts away data/hardware characteristics and provides a unified interface for data accesses. This way performance is improved while tiering complexity is hidden from the application layer.

1 INTRODUCTION

Data warehouse systems such as Teradata [14], Oracle [7] and IBM DB2 [5] have been extensively used for processing analytical queries at scale. Such systems first collect data from various sources (e.g., sensors, logs, etc.) and then apply an expensive ETL process to make it suitable for ingestion. However, ETL increases query to answer time and results in unnecessary work when queries target only a small fraction of the data set.

In contrast to warehouses, data lakes are raw data ecosystems that manage data from multiple sources and process it in-situ, avoiding the long and expensive ETL. Each data set preserves its own format and execution is optimized to deal with heterogeneity. Thus, there may be, for example, queries that combine CSV and JSON data without having to transform and persist them first into a common representation form. This allows for faster response times and easier integration of new sources.

The volume of data and the inherently distributed nature of a data lake require a decentralized architecture that involves multiple compute and storage nodes. Conceptually, data is stored in a shared pool, that is accessed over the network and is exposed to the various applications. Large cloud vendors, such as Amazon [1] and Microsoft [10] follow this approach and offer data access on top of S3 or HDFS respectively.

Nevertheless, network access introduces additional latency and misses optimization opportunities in cases where there is temporal or spatial locality in the data access pattern. Thus, the first challenge we need to deal with is to avoid unnecessary data copies while keeping the flexibility of a decoupled architecture. Moreover, we would like to cache intermediate results without having to tune every individual engine involved in the data lake.

The two above arguments mandate the need for more sophisticated storage management in data lakes. We envision a system that tracks user workloads and automatically identifies caching opportunities independently of the source of origin of each data set. Storage management should not be tightly coupled to a specific engine but equally serve them all. However, data placement has not only performance-related implications. As main memory is still a more expensive and scarce resource than hard disk drives, placement also affects available capacity and monetary cost. With the hardware advancements in storage technologies, the placement optimization problem becomes even more difficult; the choice is not anymore binary (memory or disk), but several tiers are involved (e.g., SSD, NVM), each with its own performance-capacity-cost offering. A properly designed storage manager should be aware of the trade-offs the various tiers offer and transparently move data across them based on a given policy.

Apart from being independent from the underlying data stores, storage management should be also decoupled from the application layer of the data lake. Existing storage systems expose tiering information and decision making to the user. For example, HDFS supports tiering by using the *Archival Storage* [3] and lets the user select the right tier. We argue that in a data lake, where a multitude of different users and query engines coexist, permitting each of them to define its own policy would result in a too complicated design where conflicting decisions would undermine opportunities for data-sharing and multi-engine optimization.

This work addresses the aforementioned problems and presents the preliminary design of a storage manager specifically tailored for data lakes. The proposed system is being developed as part of the *Smart Data Lake* (SDL) [12] project. SDL is a scalable, elastic and hardware-conscious data lake that supports analytical tasks.

The proposed storage manager is a self-tuning and elastic component that enables efficient data placement and access. We can think of it as a middleware between storage and compute nodes. It supports direct access to raw data but it can also create on-demand intermediate representations of various formats (e.g., CSV, columnar binary, etc.) and persist them in the appropriate tier. SDL considers several different storage alternatives, each featuring a different set of characteristics. Our storage hierarchy includes remote cloud storage, local spinning disks, SSDs, NVMs and DRAM, which depending on the use-case can be either shared between different processes or private.

In addition, the proposed architecture offloads the tiering policy from the application layer to the storage manager. This way, we enable shared optimizations and relieve the user from the burden of fine tuning storage configurations. To simplify the interaction with both the storage and compute nodes, storage manager exposes an object store-like API. Data is seamlessly exchanged in both sides (storage/compute nodes) through simple primitives, while all the complexity of the different data formats and locations is abstracted away.

The remainder of the paper is organized as follows: Section 2 gives an overview of the SDL architecture, Section 3 discusses the design decisions for the storage manager and Section 4 presents a preliminary evaluation. Finally, Section 5 concludes the paper.

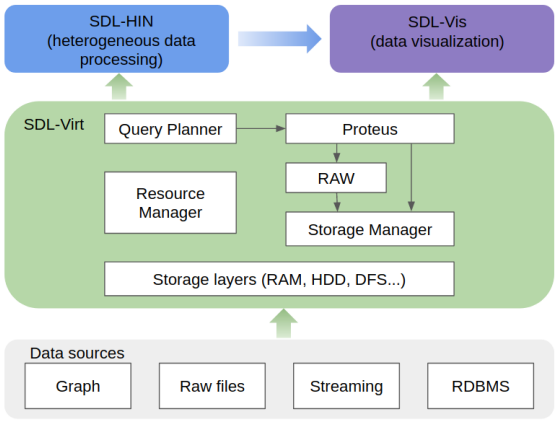

Figure 1: SDL architecture.

2 SDL ARCHITECTURE

SDL features a data lake system that supports the full stack of operations required in typical analytics tasks. Namely, it provides efficient access methods for both on premise and cloud storage, in-situ processing of diverse data, a rich set of data mining algorithms and a visualization layer that enables comprehensive data exploration. More specifically, as Figure 1 shows, SDL comprises three main components: SDL-Virt, SDL-HIN and SDL-Vis.

SDL-Virt lies at the core of the data lake. It provides all the necessary mechanisms for virtualizing data and processing it over heterogeneous hardware. Since data access always goes through SDL-Virt, it acts as a storage and query interface for the other components, that abstracts away the complexity of the various data formats and locations. For managing the ever-increasing data volume, a distributed architecture is used while the employed resources are elastically allocated to meet demand. **SDL-HIN** is an extensible suite of algorithms for the scalable analysis and mining of heterogeneous information networks [13]. It is offered as a library and runs on top of SDL-Virt. **SDL-Vis** comprises a set of tools for visually exploring unknown data sets and interpreting the results returned by SDL-HIN and SDL-Virt.

Since the SDL data lake engine is empowered by SDL-Virt, we delve into its internals and give further details about the design and the architectural choices. SDL-Virt features a distributed architecture that consists of multiple nodes that collaboratively process large data sets in a data-parallel manner. Each node runs two collocated instances of a storage and a compute node: (i) *RAW* [11] and (ii) *Proteus* [2] respectively.

RAW is a commercial system able to access and process data sets of various formats. No matter whether data resides in external infrastructures (e.g., Amazon S3, Dropbox, etc.) or in an in-house database, RAW can seamlessly access and load it through a proprietary query language.

Although RAW also provides basic processing capabilities itself, it does not take advantage of modern hardware. For the efficient execution of analytical operators, SDL-Virt employs Proteus: our just-in-time (JIT) compiled engine for fast, in-memory analytics. Performance in Proteus comes through customization and hardware heterogeneity. By JIT-ing code, we customize data access for the query/data set at-hand, and by using hybrid execution plans that involve both CPUs and GPUs, we increase parallelism and reduce end-to-end execution time.

To coordinate the execution of the RAW-Proteus nodes, SDL-Virt includes a *Query Planner* and a *Resource Manager*. As in any database system, the Query Planner parses SQL queries and generates first a logical and then a physical plan of execution. In order to do so, it takes into account hardware availability, data location and data set characteristics. Similar to YARN [16] or Mesos [6], our Resource Manager is responsible for allocating resources (e.g., CPU cores, GPUs, memory) upon query execution.

The description above implies that for every submitted query, RAW accesses a remote data set and retrieves it over the network. Such a process would severely harm performance, as it does not take into account the temporal and spatial locality of data access patterns. To deal with this issue, our design also includes a *Storage Manager*. Storage Manager detects recurring patterns in data accesses and caches data accordingly. Moreover, the plethora of available storage devices (e.g., HDD, SSD, NVM, DRAM, etc.) offers the opportunity to explore the performance-capacity-cost continuum in a more fine granular way. Each node of the consortium organizes devices in a hierarchy and data sets are moved across different storage tiers in order to maximize throughput.

Albeit storage management plays a key role in the performance of data lakes, it has not been extensively investigated yet. Existing solutions either ignore the underlying storage hierarchy or expose it to the applications and put the burden of tiering to the user. To remedy this, we opt for a transparent solution where data/hardware complexity is hidden from the user.

3 STORAGE MANAGEMENT

Storage Manager acts as a middleware between the storage and compute layers of a data lake and can serve read/write requests from both sides. In the SDL architecture, we install a storage manager instance in every node along with the Proteus and RAW processes. This way, we favor locality and reduce network traffic. Although in this work we focus on Proteus and RAW, *Storage Manager* is not coupled to them and can work with any storage/compute engine that implements its interface.

Upon query execution, a compute node contacts its local *Storage Manager* instance to get information on data location: if data is locally available, it does not need to contact RAW and directly access it from the corresponding tier. As storage devices, we consider both local (e.g., SHM, SSD, HDD etc.) and remote (e.g. HDFS, Amazon S3 etc.) resources.

3.1 Architecture

Figure 2 illustrates an overview of the architecture of a single *Storage Manager* instance. It comprises (i) an interface that exposes basic data access primitives, (ii) a lock-free message queue for the communication with Proteus and RAW, (iii) a local catalog and (iv) a tiering policy. Yellow arrows reflect data flow, while blue ones metadata flow and the interaction between the various componentns. Next, we elaborate on each of these separately.

Object store interface. For accessing data, we expose object store-like reader and writer interfaces. Object stores provide generic *get*/*put* primitives that are compatible with a wide range of data formats and query engines[1]. This way, we offer an unified interface that minimizes the complexity between data access and computation. Each dataset can be stored as a set of objects. The semantics and size of an object are arbitrary and are defined and managed by the corresponding query engines. In our architecture, objects are grouped into segments and each segment is placed in specific storage layers/tiers (we use the terms interchangeably). For allocating segments and grouping objects into segments, we have implemented special *storage allocators*, one for each tier.

[1]For simplicity, we use "query engines" to refer to both Proteus and RAW at once.

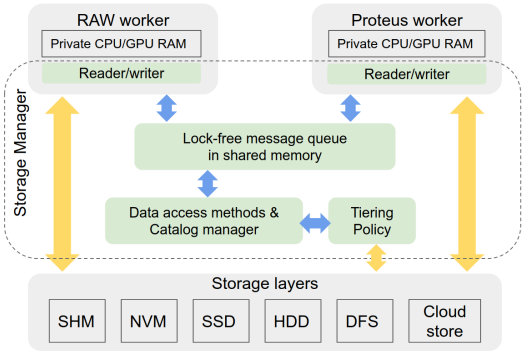

Figure 2: The anatomy of Storage Manager. Yellow arrows represent data flow while blue ones metadata flow.

Lock-free queue. As *Storage Manager* physically lives in its own process, the read/write requests for the various objects need to be exchanged via an inter-process communication (IPC) mechanism. To minimize latency, for IPC we use a queue allocated in shared memory. However, even when using such a fast IPC mechanism, the shared queue defines a critical section in the communication between the query engines and the storage manager. To reduce the synchronization cost, we design a lock-free protocol based on GCC atomics. Moreover, while the requests queue is shared, to further reduce contention, we use separate response queues for each query engine.

Another issue we need to deal with stems from the unpredictability of the arrival rate of I/O requests. Existing storage systems follow two common approaches: polled I/O and interrupt-driven I/O. The former has lower latency and higher throughput when processing high-frequency requests, while the latter consumes less CPU cycles in the case of low-frequency requests [15].

To get the best of both worlds, we support adaptive switching between the two I/O modes. Normally, *Storage Manager* works in the interrupt-driven mode, i.e., it sleeps and waits for requests. We implement interrupts with signals. After pushing a request to the queue, the corresponding reader/writer sends a signal to notify *Storage Manager*. After it has been notified, Storage Manager works in polled mode: it loops and checks for new requests. This happens for a configurable amount of time (e.g., 10*msec*) before it returns back to the interrupt mode. Thus, we use polled mode for high-frequency arrivals and interrupts for low-frequency.

Local catalog. For managing data placement, Storage Manager uses a local catalog. The catalog maps each object identifier to a unified logical address space that spans all available tiers and facilitates indexing. Each object is assigned a logical address that consists of three parts: a storage layer id (8-bits), a segment id (24-bits), and a relative offset (32-bits) within the segment. For indexing layers, we use a bit per tier and hence, we currently support up to 8 tiers. A segment id can be a pointer in case of memory-based tiers or a file descriptor for disk-based ones. The translation process between logical and physical addresses is depicted in Figure 3 and is based on a two-level lookup table. In the first step, we search by layer id in order to identify the segments that exist in the specific layer. Then, in a second step we lookup the segment id. The result of this search is the physical address of the desired segment. By adding the relative offset, we reconstruct the full physical address of the requested object.

While it is the catalog that maintains logical addresses, the translation takes place within the query engines and not the Storage Manager. Each query engine has all the layer and segment ids cached. For every I/O request, Storage Manager responds with

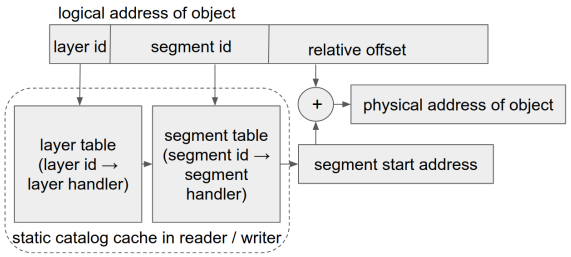

Figure 3: Translation between logical and physical addresses

the corresponding logical address and then the query engine computes the physical address and directly reads/writes data, with zero-copy and by-passing Storage Manager completely.

Tiering policy. The tiering policy is an easily plugable component that defines the placement of each object: it decides the right tier for a new object and triggers data moves, across the storage hierarchy, for existing ones. In the literature, there are many approaches for taking such decisions (e.g., heuristics [9], ML [4]). In our system, while we have implemented all the APIs and mechanisms for supporting such a policy, the design of a sophisticated data placement algorithm is part of ongoing work.

In contrast to existing solutions, we hide tiering from the application layer. We argue that this decision has several advantages. First, it simplifies implementation in the application layer. Users do not have to provision for tiering or put effort on fine-tuning storage configurations. Second, policies are re-usable and do not need to be implemented separately by each query engine. For example, Proteus and RAW do not have to implement the same data placement algorithm. Last but not least, by taking decisions based on the global access patterns we can support multi-engine optimizations. Data that is often accessed by multiple query engines is automatically pushed to the top of the storage hierarchy.

To identify interesting access patterns, we should also have in place a workload tracking mechanism. The tracking mechanism is also part of the *Tiering Policy* module. The more fine granular tracking is, the more elaborate decisions we can take. For example, consider a relational table where we only access a single column. In that case, we need to only cache this column and not the table. Although our example focuses on columns, in general the data container of interest depends on the data format. However, our object-store design is format-agnostic and of size that depends on the application. Therefore, an object can be anything from a single tabular record to a whole graph data set.

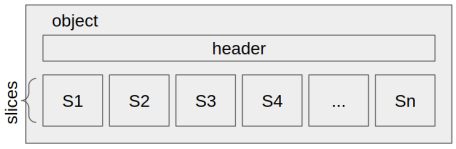

Figure 4: Data object structure

To solve this problem and provide fine-granular tracking, we devise *slices*. Each object comprises multiple slices of fixed size as depicted in Figure 4. Slices are not exposed through Storage Manager's API and readers/writers are unaware of their existence. As we use buffered I/O, each time a buffer flushes, a new slice is transparently created. In the example of Figure 4, let us assume that the object is persisted in a SSD drive and that slices *S2*, *S4* are frequently accessed. Then, the tiering policy is responsible to create a new object containing only *S2* and *S4* and store it in a higher-level tier (e.g., DRAM). Please note that this happens independently of the data semantics and format.

Table 1: IPC max throughput (messages per ms)

shared queue	sockets	pipes	System V mq
2406.5	186.6	150.8	365.8

3.2 Read/Write Workflow

Having described all the components of Storage Manager's architecture, for better understanding how they work, we now describe the workflows for the write and read access paths.

Write. Let us assume that a query/storage engine (e.g.,RAW) sends a write request to the storage manager for persisting an object with unique identifier X. The request is sent through the lock-free shared queue. Storage Manager receives the request and based on the tiering policy decides on the tier that the object should be placed (e.g., SSD). Then, the manager contacts the storage allocator of the specific tier, in order to get a segment id and the position of the new object within the chosen segment. Having a tier identifier, a segment id and an offset within the segment, we can form a logical address L and Storage Manager persists the mapping $X \rightarrow L$ to its catalog. Finally, the manager returns the logical address to the engine that issued the request.

Read. Upon a read request, Storage Manager checks the catalog and retrieves the logical address for the specific object id and returns it to the query engine in order to read the object from the corresponding tier, without waiting for data copying around tiers. Hardware-conscious systems (e.g., Proteus) need to prefetch data into a privately managed CPU/GPU memory in order to unleash their full potential. To enable this behavior, our Storage Manager provides two additional operations: *load* and *unload*. The first one prefetches data while the second one evicts objects from the private memories , which are not managed by Storage Manager. Thus these APIs are not against our overall goal of storage virtualization.

4 EXPERIMENTAL EVALUATION

We perform a set of microbenchmarks to evaluate our design decisions for serving I/O requests and accessing data. More specifically, we assess the proposed IPC mechanism (lock-free queue) and the importance of zero-copying while accessing data. As this is a primitive version of our system, and tiering policies are not yet in place, we leave end-to-end evaluation for future work.

The experimental setup is a 2-socket server with Intel Xeon E5-2650L v3 CPU 1.80GHz, 24 threads/socket and 256GB DRAM. In these experiments, all data objects are cached in memory.

Lock-free shared queue. We compare, in terms of throughput, our lock-free message queue with other IPC methods by sending 8-byte sized 10 million packets. The results are shown in Table 1. In case of IPC between two processes, we achieve 6.6× the throughput of a *System V mq* [8].

We also assess the impact of adaptive I/O when processing queue requests. We send $1K$ read requests to the queue as fast as possible and measure response latency and CPU utilization. When in polled I/O mode, the response latency is $6.4\mu sec$ and CPU usage 100%; we need an entire hardware thread just to poll the queue. When using interrupts, CPU utilization drops near to 0%, but we get a 5.7× latency ($36.5\mu sec$). By switching between the two modes, we ensure both low latency and resource efficiency.

Zero-copy. IPC mechanisms involve from zero to multiple data copies. Techniques that buffer data in kernel space (e.g., pipes, sockets) usually need two copies, while shared memory-based approaches require one or no copies at all. Reading data from Storage Manager follows the zero-copy approach. To quantify the potential gain in data access latency, we conduct the following experiment: a process reads $10MB$ of data from Storage Manager and we measure the elapsed time between sending the request and starting to consume data at the reader's side. To simulate the various IPC methods, we artificially add extra rounds of copying on top of our zero-copy mechanism. Moreover, we compare against *grpc*: a widely used protocol based on http and protobufs. Results are shown in Table 2. We observe that even a single copy increases access latency by 3 orders of magnitude.

Table 2: Data access latency for various IPC protocols

protocol	0-copy	1 copy	2 copies	grpc
Avg. latency (us)	6.6	1996.8	3478.0	20790.2

5 CONCLUSION

In this paper, we present a storage management solution especially tailored for data lakes and build the proposed system on top of the *Smart Data Lake* platform. Our design hides the data/hardware complexity of a data lake and provides unified and transparent access to different tiers of the storage hierarchy. Although all the mechanisms that enable the proposed idea are in place, we still lack sophisticated tiering policies. Our future plans focus on this aspect, i.e., algorithms that lift workload characteristics in order to better exploit the underlying storage.

ACKNOWLEDGMENTS

This work has been funded by the European Union's Horizon 2020 research and innovation programme under the grant agreement No 825041 (SmartDataLake).

REFERENCES

[1] Amazon. 2020. *Data Lake on AWS.* Retrieved Dec20, 2020 from https://aws.amazon.com/solutions/implementations/data-lake-solution
[2] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines.* Technical Report.
[3] The Apache Software Foundation. 2020. *Archival Storage, SSD and Memory on HDFS.* Retrieved Dec20, 2020 from http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html
[4] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. In *MSST*.
[5] IBM. 2020. *IBM Db2 Warehouse.* Retrieved Dec20, 2020 from https://www.ibm.com/products/db2-warehouse
[6] Apache Mesos. [n.d.]. . http://mesos.apache.org/
[7] Oracle. 2020. *Oracle Autonomous Data Warehouse.* https://www.oracle.com/in/database/technologies/datawarehouse-bigdata.html
[8] Linux Manual Page. 2020. *MQ Overview.* https://man7.org/linux/man-pages/man7/mq_overview.7.html
[9] S. Qiu and A. L. Narasimha Reddy. 2013. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *MSST*.
[10] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 51–63.
[11] RAW-labs. [n.d.]. . https://www.raw-labs.com/
[12] SmartDataLake. 2020. *Sustainable DataLakes for Extreme-Scale Analytics.* Retrieved Dec20, 2020 from https://smartdatalake.eu
[13] Yizhou Sun and Jiawei Han. 2013. Mining heterogeneous information networks: a structural analysis approach. *Acm Sigkdd Explorations Newsletter* 14, 2 (2013), 20–28.
[14] Teradata. 2020. *Teradata Integrated Data Warehouses.* Retrieved Dec20, 2020 from https://www.teradata.com/Products/Software/Integrated-Data-Warehouses
[15] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll is Better than Interrupt. In *FAST*.
[16] Apache YARN. [n.d.]. . https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html