CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D3.1 Early release of Ephemeral Data Store

Due date of deliverable: 30-06-2023
Actual submission date: 30-06-2023

Start date of project: 01-01-2023 Duration: 36 months

Summary of the document

Document Type	Other
Dissemination level	Public
State	v1.0
Number of pages	12
WP/Task related to this document	WP3 / T3.1
WP/Task responsible	IBM
Leader	Bernard Metzler (IBM)
Technical Manager	Raúl Gracia (DELL)
Quality Manager	Marc Sanchez-Artigas (URV)
Author(s)	Bernard Metzler + Pascal Spörri (IBM)
Partner(s) Contributing	IBM, DELL, URV
Document ID	CloudSkin_D3.1_Public.pdf
Abstract	Early software release of the data store, focusing on definition of API's, data store architecture and application integration of the performance tier.
Keywords	Key-value store, ephemeral data, persistency, elasticity

History of changes

Version	Date	Author	Summary of changes
0.1	27-04-2023	Bernard Metzler, Pascal Spörri	First draft.
0.2	11-05-2023	Pascal Spörri	Fixes, added Python integration.
0.3	21-06-2023	Raúl Gracia, Bernard Metzler	Adressed Raúl's comments.
0.4	22-06-2023	Bernard Metzler	Added section onsupporting ephemeral tasks.
1.0	30-06-2023	Bernard Metzler, Pascal Spörri	Final version.

Table of Contents

List of Abbreviations and Acronyms

API	Application Programming Interface
CC	Creative Commons
CSV	Comma-separated values
CUDA	Compute Unified Device Architecture
DOI	Digital Object Identifier
GEDS	Generic Ephemeral Data Store
IO	Input/Output
JNI	Java Native Interface
KV	Key/Value
MDS	Metadata Service
NVMe	NonVolatile Memory express
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call
TCP	Transmission Control Protocol

1 Executive summary

In distributed systems, the efficient handling of temporary data between tasks of a complex, potentially multi-staged workload is critical to the end-to-end execution performance. The heterogeneity of access patterns, object sizes, lifetimes, locations and persistency requirements of those ephemeral data across the diversity of applications often lead to the implementation of application specific, integrated handling of those data. In those cases, integrating with off-the-shelf storage platforms, such as established key-value stores or distributed file systems would simply miss the desired overall application execution performance targets. Here, maintaining overall performance is preferred over generality of the solution.

With the **G**eneric **E**phemeral **D**ata **S**tore (GEDS) we try to tackle this storage solution pandemonium. We are starting out with lessons learned during the design of another high-performance ephemeral data store, *Crail* [1], which exemplifies a design for high performance, distributed object access and flexible storage multi-tiering. While starting out with a complete new data store design, we add to it two goals, which were not fulfilled by *Crail*, namely deep integration of application buffers into the storage hierarchy and the support of task synchronisation primitives. The first such primitive which will be supported by GEDS is a pub/sub service. It will allow clients to register for events signaling data objects to become available or changed.

We aim at using GEDS as a generic ephemeral data store within the CloudSkin project, with a focus on the specific requirements of a data store for serverless workload execution environments in a Cloud core-edge continuum.

As of the time of this first milestone writeup, after starting from scratch, we now maintain a stable prototype of the GEDS data store with limited functionality, which is openly available at [2]. It comprises a base implementation of the high performance lowest storage tier, which directly integrates with the application task as a dynamically loadable library. It further implements access to a persistence tier via AWS S3 interface. The core of GEDS is written in C++. It exports native language bindings for Java via JNI and for Python via *PyBind11* [3].

To enable integration into existing projects, we created plugins for common libraries. For Java-based workloads we built a file-system plugin for *Apache HDFS* [4]. For Python-based workloads we integrated with `smart_open` [5]. Both libraries are used to access files on both local and remote filesystems.

2 Data store requirements for CloudSkin

This section summarizes the requirements on the ephemeral data store as to be deployed in the envisioned Cloud-edge continuum. The specific design of GEDS is derived from those requirements.

- **Performance:** In distributed systems, the efficient handling of ephemeral data is critical for the overall performance of workload execution. While resource virtualization is inevitable as a concept, maintaining close to bare metal infrastructure performance at the storage level is needed to enable the flexible and fine granular decomposition of complex application workflows into sets of parallel and short-lived tasks, so that the CloudSkin platform can efficiently run its intended use cases in a Cloud-edge continuum.
- **Elasticity:** In a dynamic edge-core continuum, data store elasticity enables spending only storage resources which are currently needed to hold working set data. It distinguishes it from a classic managed service (e.g. a key value store), which, for example, cannot scale down to zero. Elasticity is crucial for overall resource efficiency.
- **Multi-tiering:** While ephemeral data are typically stored in DRAM memory, extended data tiering can be beneficial. (1) It allows to dynamically exceed local storage resource bounds, accidentally or intended (relocating less frequently used objects for very thin compute nodes), (2) it enables the integration of an object store for reading input data and writing final output data seamlessly within a common namespace, and (3) a 'higher' storage tier can be used for checkpointing and recovery of working set data in case of failure.
- **Object location awareness:** This feature, at one hand, goes along with the performance requirement that maintaining data as close as possible with the current task avoids network data transfers, as well as local data copy operation during read and write access. Furthermore, combined with task scheduling it can enable the coordination of data output placement and compute task scheduling, aiming at data and compute colocation at the same node.
- **Additional services:** The GEDS data store will support a unified object namespace for all tasks and stages of a distributed workload. We plan to extend the services of this ephemeral namespace from just maintaining access to data objects with the support of simple communication primitives, involving maintained data objects. As a first additional service, it is planned to provide a pub/sub service, which allows data store clients to register with future data object creation or changes to already existent objects. We expect this service to be useful for the synchronisation of successive tasks within a workflow.

3 GEDS Design

This section introduces the overall desgin of the GEDS data store. At the time of this writing, the main design decisions have been made. However, we reserve the right to make further changes and additions to it.

3.1 Overall Design – High-Level View

3.1.1 Components

Figure 1 depicts the overall design of the Generic Ephemeral Data Store. The data store mainly comprises the following components: Data storage tiers, clients and a Metadata Service.

We have defined three distinct storage tiers: *Tier Zero* implements a high performance storage tier, where data objects reside in local node memory, potentially being memory mapped into application buffers. *Tier 1* represents a disaggregated storage tier designed to keep data objects in high performance block storage, preferable NVMe disks. The *Persistence Tier* is the lowest storage tier, where data can be persisted into and retrieved from an available KV-store.

Each GEDS instance runs a *Metadata Service* (MDS), which is synchronising object creation, access and location. The current prototype maintains one MDS instance per GEDS. Scaling out to multiple

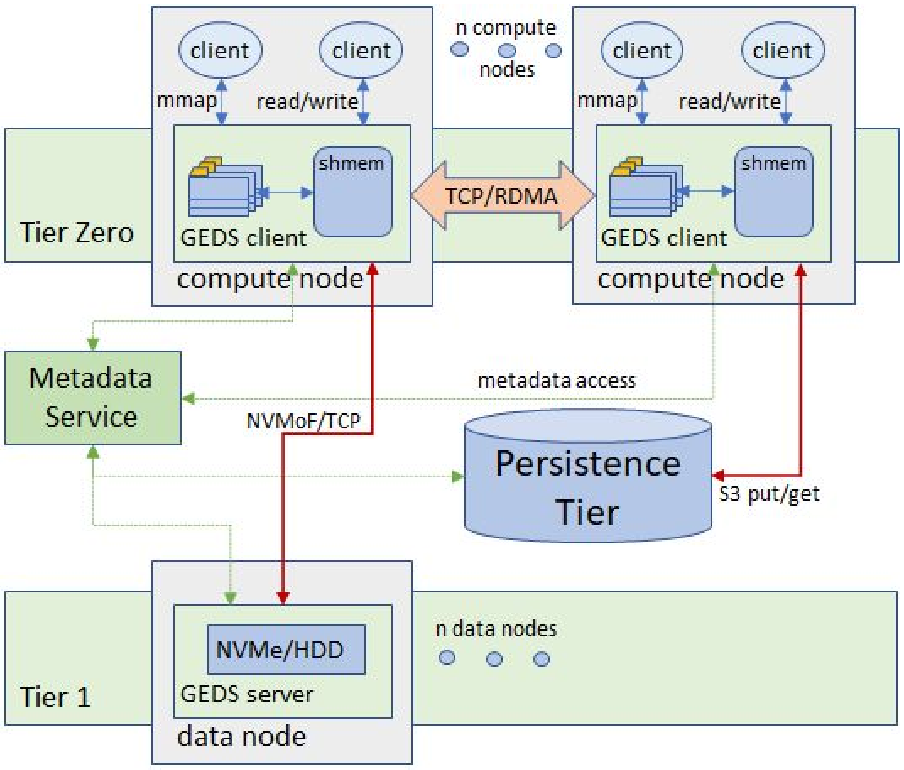


Figure 1: GEDS Overall Design

instances or to a managed service like *Redis* is possible, but at the current stage of the project not needed.

Clients to the data store can join the GEDS service by loading a `libgeds` client shared library. After loading, the client registers itself with the MDS and then may passively (reading data objects) and actively (registering and writing data objets) participate in the GEDS service.

3.1.2 Data object handling

GEDS establishes a shared namespace for data objects. Data objects are uniquely named by an identifier or *object key*, following naming conventions very similar to those of the AWS S3 (Simple Storage Service), see [6].

A new data object enters the GEDS namespace either after reading it in from the Persistence Tier, or after registering a local application buffer as a GEDS object. With that, each data object starts its lifetime within GEDS Tier Zero. As a design principle, any access to a data object will instantiate that object at Tier Zero of the local node the client is running on. This may cause several local copies of the same object if that object is accessed from clients on multiple nodes. Data copies are tracked by the MDS.

Within Tier Zero, data transfer among nodes happens directly between application buffers. For efficiency reasons, GEDS currently implements its own TCP based RPC communication library for object transfers between nodes. It is considered extending it for RDMA based transfers, if high performance network infrastructure is available (see Figure 1).

3.2 Efficient application integration

The GEDS client operations are consolidated within a shared library. Loading the library will connect the client with the MDS and let it become part of the GEDS namespace, since implementing a local share of Tier Zero. Only during remote object access, the library will automatically connect with the peer currently hosting that data object. GEDS keeps track of any files opened by the application. If a file is closed then GEDS is able to release its associated resources.

3.2.1 Zero-copy data object access

Efficient integration of data object access with the data-store is one of the major design points of GEDS. Therefore, Tier Zero of GEDS implements support for zero-copy memory mapped data access.

3.2.2 Python Integration

GEDS integration with Python is achieved via Pybind11 [3]. PyBind11 along with GEDS and its dependencies are built into a separate dynamic library `pygeds.so`.

Python libraries are built on top of `pygeds.so`. To ease integration with scientific applications we built a plugin for [5], see Figure 2.

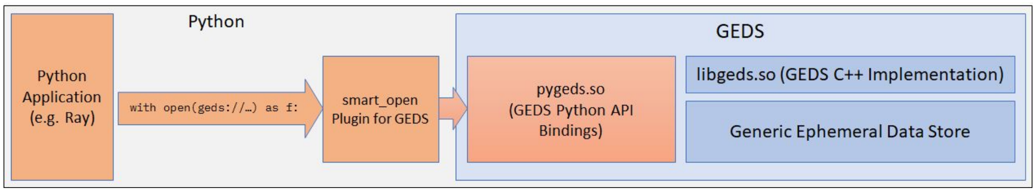
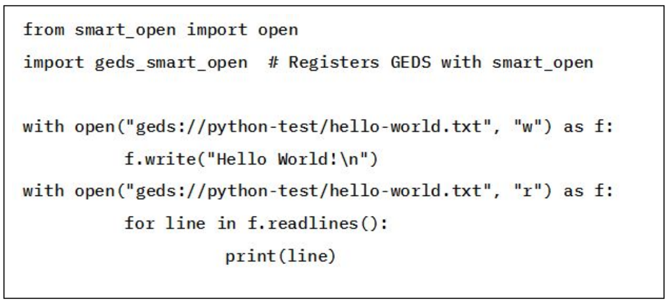

Python
Python Application (e.g. Ray)
with open(geds://…) as f:
smart_open Plugin for GEDS
GEDS
pygeds.so (GEDS Python API Bindings)
libgeds.so (GEDS C++ Implementation)
Generic Ephemeral Data Store

Figure 2: Python bindings for GEDS

Figure 3 exemplifies `smart_open` based client side Python integration with GEDS.



```
from smart_open import open
import geds_smart_open  # Registers GEDS with smart_open

with open("geds://python-test/hello-world.txt", "w") as f:
          f.write("Hello World!\n")
with open("geds://python-test/hello-world.txt", "r") as f:
          for line in f.readlines():
                    print(line)
```

Figure 3: Using GEDS services from Python

In its current state the Python integration does not expose any memory-mapped files. Reads and writes to objects are directly handled in GEDS. We plan to integrate support to integrate memory-mapped files directly to the Python application.

3.2.3 Java Integration

GEDS provides the HDFS file system interface to allow for seamless integration with Java based applications. Similar to the Python library we build a separate library for Java. We expose the same set of API bindings via JNI to Java. We then wrap the native bindings with a separate set of Java Classes to enable easy access to GEDS functionalities. Both `geds_java.so` and `geds.jar` containing the dynamic library for Java and the GEDS bindings are shipped as dynamically compiled libraries.

A very common interface for accessing files is the Apache HDFS Filesystem library [4]. It is the standard way to access files on both local and remote filesystems for Apache Spark [7] and many other Java-based distributed computing frameworks.

GEDS implements a HDFS library that builds directly on the previously discussed Java bindings. The `GEDS-HDFS` library implements input and output streams that can be consumed directly by any Java application. It also wraps all the filesystem operations and maps them to GEDS (see [8] for the current implementation of `GEDS-HDFS`). Figure 4 exemplifies a use case, where GEDS is deployed as an ephemeral data store for Apache Spark.

3.3 Multi-tiering and object persistency

The definition of three storage tiers is mainly driven by the following goals:

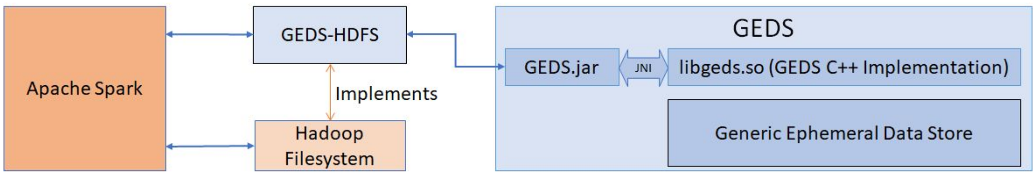

Apache Spark
GEDS-HDFS
Implements
Hadoop Filesystem
GEDS
GEDS.jar
JNI
libgeds.so (GEDS C++ Implementation)
Generic Ephemeral Data Store

Figure 4: Java integration with example Apache Spark application.

- **Selectable persistency**: Data objects residing only at Tier Zero are completely ephemeral. Any node crash of a non-replicated object will make it unavailable. Tier 1 and Persistence Tier introduce different levels of data persistency. Data objects can be replicated to these two tiers to achieve persistency. Checkpointing working set data to a lower tier might be a typical use case. At the current stage of GEDS design and implementation, the application interface for selecting a certain level of object persistence is still undefined.
- **Elasticity and scalability**: The amount of data locally maintained at a node's Tier Zero may be limited by the node's resources. Automatically spilling out less used objects to a lower storage tier can mediate this situation under storage pressure. Spilling to Tier 1 is expected to have less performance impact than spilling to the Persistence Tier.

  Tier 1 will serve as a high performance storage tier, exceeding the performance of a remote KV store (Persistence Tier). Its disaggregated implementation further helps to scale compute and storage resources independently.
- **Input/output integration**: Typically, input and output of data intense workloads will be at a KV store such as S3. Integrating that resource into the namespace of GEDS enables the generation of seamless workflows. Logically, data reside in one data store during all stages of a workload, while physically moving across storage tiers.

3.4 Elasticity

GEDS tries to keep storage resource consumption at its minimum. At Tier Zero, it does not require the pre-allocation of any node memory before data objects are locally instantiated. Data objects are instantiated with the local nodes file system (currently using `/tmp`) and, depending on the application, are memory mapped for zero copy application access. With that approach, a node (1) never consumes more local memory than the aggregated size of locally instantiated objects, and (2) a node's virtual memory may keep only parts of the data objects in physical memory under memory pressure. Implicitly, there is no need to pre-allocate any storage resources with a compute node participating in the GEDS namespace. Any level of current storage resource usage heterogeneity among compute nodes is solely driven by current application data processing. This puts GEDS in stark contrast to other high-performance ephemeral data store implementations, such as the Plasma data store integration, where all compute nodes are started with the same, pre-defined amount of DRAM resources exclusively dedicated to the Plasma store [9].

It is planned to implement resource elasticity for Tier 1 as well, while it comes natural for using a KV store based Persistence Tier.

3.5 Data object location awareness, steering and relocation

As provided by the MDS, the GEDS namespace describes each object by a unique string, the current tier it is located in, and the host address of the node where it is located in case of Tier Zero. With that, GEDS will enable the controlled co-location of compute and data objects at Tier Zero. Combined with appropriate control over task/function scheduling policies, physical data transfers during job execution can be minimized.

Another functionality enabled by data relocation within Tier 0 is the ability to decomission compute entities, which do not participate in current workload execution, but still hold data objects from

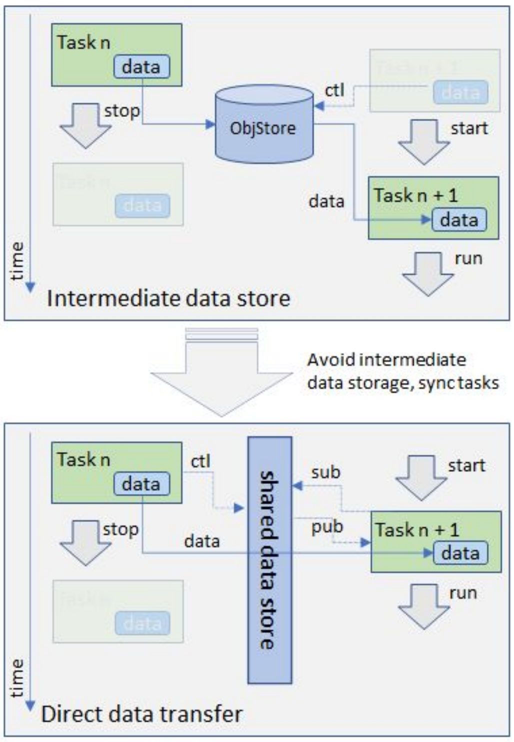


Figure 5: Using pub/sub service for efficient task synchronization.

previous activities. Those data objects can be either relocated within Tier Zero or spilled out to a lower storage tier.

Relocating data objects of course also serves as the mechanism to achieve data persistency (see Section 3.3).

3.5.1 Supporting ephemeral tasks in a Cloud-edge scenario

In a Cloud-edge scenario, edge compute tasks may be ephemeral, but interact with the Cloud core. GEDS provides mechanisms to provide this continuum by:

- Providing a shared object namespace covering edge and core: Irrespective of its current location, data objects remain accessible for all compute entities involved.
- Enabling efficient data relocation: A compute entity, such as an edge device, may execute a completely ephemeral task, disappearing after task execution, while the tasks data output is relocated within the shared GEDS namespace, to be picked up by a followup task (see also Figure 5).

3.6 Additional services

We plan to extend the functionality of GEDS beyond maintaining ephemeral data objects. At the current stage of the project, we are investigating the suitability of adding a *pub/sub-service*, which allows a client to register with the MDS for updates on the status of GEDS data objects. It will allow triggering of client activities related to subscribed objects, such as the consumption of data objects which are the result of a previous stage of data processing, avoiding the management of an explicit control flow among stages, also enabling direct object transfers between consecutive tasks (see Figure 5).

4 Status of the GEDS prototype

We integrated GEDS into Apache Spark benchmark for TeraSort and TPC-DS [10]. GEDS acts as a ephemeral data-store for Spark shuffle, or as a transparent cache for the input data located on AWS

S3 or IBM COS.

Python applications support the simple `smart_open` interface for reading and writing files. Multi-processing with Python requires communication over TCP: GEDS does not support sharing its state on the local host yet.

4.1 Overall functionality

Table 1 depicts the overall implementation state of GEDS.

Table 1: State of GEDS

Feature	State	Comment
Basic Features		
Create buckets	Done	
List buckets	Done	
Create objects	Done	
Update objects	Done	
Delete objects	Done	
List objects	Done	
Read from object	Done	POSIX-like read interface
Write to object	Done	POSIX-like write interface
Move objects	Needs work	In its current implementation objects are first copied and then finally deleted.
Copy objects	Needs work	Objects are currently copied. Copy-on-write semantics are desirable.
Advanced Features		
File metadata support	Needs work	Allows attaching metadata to objects.
Memory-mapped file access	Needs work	Compile-time flag. Only works for local objects. Requires additional semantics and design work.
Object store integration	Needs work	Object stores can be configured on a per-bucket basis. Objects can be read and written. Semantics for reading/writing needs to be investigated.
Object spilling	Needs work	First draft has been implemented. Requires a configured object store.
Pub/Sub Support	Needs work	Pull request available but not yet integrated.
Object Caching	Needs work	Only objects stored on S3 are cached in GEDS.
Object Replication	Not started	Required for memory-mapping of remote objects.
Configurable resiliency	Not started	E.g. force write-back to S3 if a file is written.
Multi-process support	Not started	One GEDS instance is created per process.
NVMeOF Support	Not started	
Library Integration		
Java integration	Done	HDFS input streams do not support ByteBuffers.
Python integration	Needs work	Not scalable since a new instance is created for each fork. No data sharing between individual processes.
Simplified wrapper	Not started	`GEDS.h` needs to be wrapped and re-exported with no external dependencies for seamless linking with C#, Go or Rust.
NVIDIA CUDA integration	Not started	Expose GEDS objects in CUDA.
Fuse Filesystem	Not started	

4.2 IO Benchmark

GEDS implements a simple IO benchmark to understand the IO read performance of remote objects. In Figure 7 we see throughput to access an object stored on a remote GEDS instance.

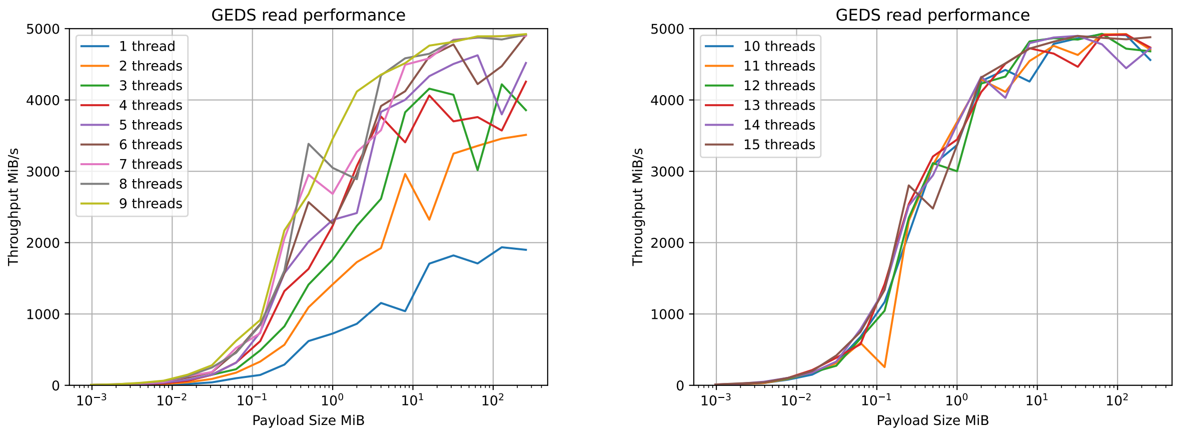


Figure 6: GEDS Read Performance. Two GEDS clients are connected over a 40gbit/s network link.

4.3 Spark shuffle for single TPC-DS Queries

We ported the dis-aggregated Spark-S3-Shuffle plugin [11] to GEDS. This plugin directly integrates into Apache Spark and is thus not able to serve per-client shuffle files. Thus we were required to use the default shuffle algorithm which suffers from reading small block-sizes. We mitigate most of these issues by asynchronously pre-fetching individual shuffle blocks, and by storing the index files in the GEDS metadata service. In Figure 7 we show the performance of individual TPC-DS queries on the IBM CodeEngine. GEDS supports spilling to an Object Store specifying a quota.

5 Conclusions and next steps

At the current stage of the CloudSkin project, a stable design of the GEDS data store has been achieved. We started with implementing a GEDS prototype with limited functionality, which already achieved decent stability and performance. It is available as an active open source project [2, 8] under Apache 2 License terms.

In particular, a Tier Zero implementation with limited functionality is available. Furthermore, we started to implement the integration with the Persistency Tier. As of now, data objects can be loaded from a KV store implementing the S3 interface and written out to it. The implementation of the Tier 1 has not yet started. We expect to be able to resemble its main intended functionality –spillover and limited persistency– by the Persistency Tier with lower performance.

There remains substantial work to be done to achieve scalability for Python integration, aiming at avoiding per-Python-Task replication of full GEDS client code on the same node.

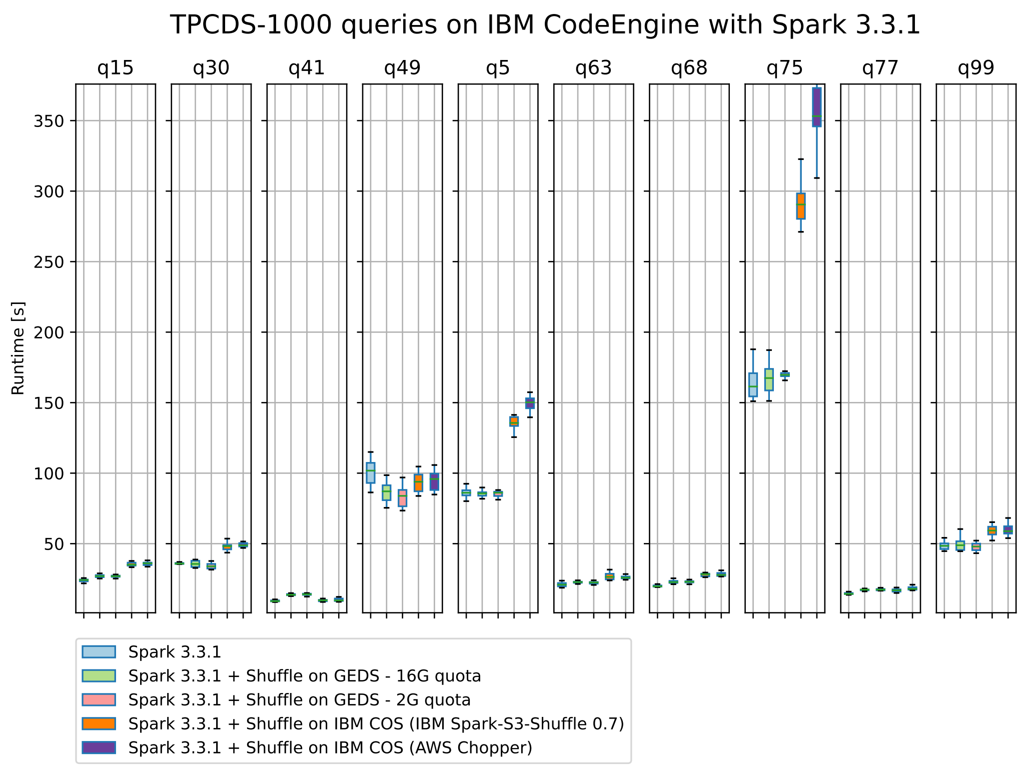


Figure 7: TPC-DS performance for single queries. We compare the GEDS shuffle plugin with [11] and [12].

References

[1] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of temporary storage in the NodeKernel architecture.," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.

[2] "Hadoop filesystem implementation for GEDS." `https://github.com/IBM/GEDS`.

[3] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between C++11 and Python." `https://github.com/pybind/pybind11`, 2017.

[4] "The Hadoop FileSystem API definition." `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/filesystem/index.html`.

[5] R. Řehůřek, "smart_open – utils for streaming large files in Python." `https://github.com/RaRe-Technologies/smart_open`, 2015.

[6] `https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html`.

[7] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., "Apache spark: a unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.

[8] `https://github.com/IBM/GEDS-HDFS`.

[9] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging {AI} applications," in 13th USENIX Symposium on Operating Systems Design and Implementation, 2018.

[10] R. O. Nambiar and M. Poess, "The making of tpc-ds," in Proceedings of the 32nd International Conference on Very Large Data Bases, 2006.

[11] P. Spörri, "Shuffle plugin for Apache Spark and S3 compatible storage services." `https://github.com/IBM/spark-s3-shuffle`, 2022.

[12] "Cloud shuffle storage plugin for apache spark." `https://docs.aws.amazon.com/glue/latest/dg/cloud-shuffle-storage-plugin.html`, 2022.