Percipient StorAGe for Exascale Data Centric Computing

FETHPC1 - 671500

WP 4 Programming Models and Data Analytics for Applications

D4.2 Analysis Framework Requirements Specification

D4.2 Analysis_Framework_Requirements_Specification_1.0

Scheduled Delivery: 01.06.2016
Actual Delivery: 30.05.2016
Version 1.0

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 671500

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission)</td>
</tr>
</tbody>
</table>

X
Revision history:

<table>
<thead>
<tr>
<th>Date</th>
<th>Editor</th>
<th>Status</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.2016</td>
<td>Clemens Lutz</td>
<td>Draft</td>
<td>0.1</td>
<td>Document Structure</td>
</tr>
<tr>
<td>2.5.2016</td>
<td>Clemens Lutz</td>
<td>Draft</td>
<td>0.2</td>
<td>Document Trunk</td>
</tr>
<tr>
<td>7.5.2016</td>
<td>Sai Narasimhamurthy</td>
<td>Draft</td>
<td>0.3</td>
<td>Added description of Mero and Clovis</td>
</tr>
<tr>
<td>8.5.2016</td>
<td>Sebastian Breß</td>
<td>Draft</td>
<td>0.4</td>
<td>Review and revision inside group</td>
</tr>
<tr>
<td>9.5.2016</td>
<td>Clemens Lutz</td>
<td>Draft</td>
<td>0.5</td>
<td>Formatting &amp; Internal reviewer</td>
</tr>
<tr>
<td>12.5.2016</td>
<td>Clemens Lutz</td>
<td>Draft</td>
<td>0.6</td>
<td>Integrated reviewer comments</td>
</tr>
<tr>
<td>23.5.2016</td>
<td>Clemens Lutz</td>
<td>Final</td>
<td>0.7</td>
<td>Integrated comments WP4 leader</td>
</tr>
</tbody>
</table>

Authors
Sebastian Breß (DFKI), Clemens Lutz (DFKI)

Internal Reviewers
Sining Wu

Copyright
This report is © by DFKI and other members of the SAGE Consortium 2015-2018. Its duplication is allowed only in the integral form for anyone’s personal use and for the purposes of research or education.

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

More information
The most recent version of this document and all other public deliverables of SAGE can be found at http://www.SAGE-project.eu
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>BDEC</td>
<td>Big Data and Extreme-scale Computing</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>D</td>
<td>Deliverable</td>
</tr>
<tr>
<td>DRS</td>
<td>Document Review Sheet</td>
</tr>
<tr>
<td>EC</td>
<td>European Commission</td>
</tr>
<tr>
<td>HDFS</td>
<td>(Apache) Hadoop File System</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>IOPS</td>
<td>Input / Output Operations Per Second</td>
</tr>
<tr>
<td>IR</td>
<td>Incremental Result</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java DataBase Connectivity (API)</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>PACT</td>
<td>PArallelization ContracT</td>
</tr>
<tr>
<td>PM</td>
<td>Project Manager</td>
</tr>
<tr>
<td>PO</td>
<td>Project Officer</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>S3</td>
<td>(Amazon) Simple Storage Service</td>
</tr>
<tr>
<td>SAGE</td>
<td>Percipient StorAGe for Exascale Data Centric Computing</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>WP</td>
<td>Work Package</td>
</tr>
<tr>
<td>YARN</td>
<td>(Apache Hadoop) Yet Another Resource Negotiator</td>
</tr>
</tbody>
</table>
Table of Contents
1. Executive Summary ........................................................................................................... 5
2. Introduction ......................................................................................................................... 6
3. Apache Flink Overview ....................................................................................................... 8
4. SAGE Platform .................................................................................................................... 12
5. Apache Flink on SAGE ....................................................................................................... 14
6. Conclusions and Next Steps .............................................................................................. 24

References ............................................................................................................................... 25

List of Figures
Figure 1: Architecture of Apache Flink ................................................................................... 8
Figure 2: Sage System Hardware ............................................................................................ 12
Figure 3 Compute Storage Mode ............................................................................................ 18
Figure 4 Offload Mode ............................................................................................................ 19
Figure 5 Full Data-Centric Mode .......................................................................................... 20
1. Executive Summary

This deliverable specifies the requirements on the Apache Flink data analytics and workflow framework to deal with data in exascale, which will be integrated with the SAGE platform by Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI). The Flink data analytics framework will embed SAGE as a data storage platform as a foundation to scale the wide range of applications built on top of Flink. We systematically analyze the interfaces and architectures of Flink and SAGE to identify the components of Flink that have to be created or modified to efficiently exploit the multi-tiered storage system. We devise a set of key challenges that must be addressed in the project, and from these derive the requirements on the data analysis framework Flink.

Concretely, the requirements on the data analysis framework Flink are high-performance storage connection to Mero, in-storage computation, multi-tiered storage, and operator spilling. The components that we have identified for creation or modification are the storage connector, runtime system, scheduler, public API, resource manager, and Mero API.

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contribution in Person Months</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFKI</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1 Partner Contributions for Deliverable
2. Introduction

We describe the positioning of this document within the SAGE project. First we define the objectives of this document. Second, we associate this document with tasks from the deliverable. Finally, we set a context for this document within related deliverables.

2.1. Objectives

The objectives of this document are as follows:

- Introduce the Apache Flink data analytics and workflow framework as a platform for large-scale, data-intensive streaming and batch processing applications.
- Document the components of Apache Flink that have to be adapted for integration with the SAGE hierarchical storage platform.
- Describe the technical challenges determined in the analysis of the interfaces and components to be adapted that must be addressed in the Flink-SAGE integration project.
- Describe the architectural design choices that achieve integration with SAGE storage system at the application modes of operation specified by SAGE.
- Document the requirements on the Apache Flink data analytics and workflow framework to integrate with the SAGE multi-tier storage platform.

2.2. Tasks associated with the deliverable

<table>
<thead>
<tr>
<th>Task Number</th>
<th>Description of Task</th>
<th>Planned activity within task for this Deliverable</th>
<th>Performed activity and any deviation from Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4.4a</td>
<td>Requirement Analysis and Definition</td>
<td>Requirement Analysis and Definition</td>
<td>Studied interfaces of Flink and Mero, studied internals of Flink, report writing</td>
</tr>
<tr>
<td>T4.4b</td>
<td>System and Software Design</td>
<td></td>
<td>Studied different architectural designs and derived requirements</td>
</tr>
</tbody>
</table>

Table 2 Tasks associated with the Deliverable
2.3. Relation to other Tasks and WPs

This deliverable formulates the requirements on the Apache Flink framework for exascale computation using the SAGE storage platform. The aim is to guide the system design and integration process by evaluating the required changes for SAGE integration and deriving possible architectural designs that will take place in D4.3 and D4.4. The workload characterization in WP1 documents the behavior of Flink applications in regard to their storage accesses, which will have to be considered when defining the design goals in the later WP4 tasks and deliverables. The percipient storage architecture is specified in WP2, which influences the design space and challenges of the Flink-SAGE integration project.
3. Apache Flink Overview

This section lays a foundation for discussions about Apache Flink in the rest of this document. First we describe Flink’s architecture at a high level. Second, we provide a view on how programs are built using Flink in both the batch and streaming contexts. Third, we give a general insight into the inner workings of Flink when a program interacts with storage.

3.1 Architecture

![Figure 1: Architecture of Apache Flink](image)

_Figure 1_ provides an overview of the Apache Flink [1, 2] architecture. The foundation of Flink is a unified runtime environment in which all programs are executed. Programs in Flink are structured as directed graphs (JobGraphs) of parallelized operations that can further contain iterations [3]. A JobGraph consists of nodes and edges. There are two classes of nodes: (stateful) operators, and (logical) intermediate results (IRs). When running a program in Flink, operators are translated into various parallel entities, which consequently process partitions of intermediate results (or input files), offering data parallelism. Unlike Hadoop, programs in Flink are not divided into individual phases that are executed sequentially (Map and Reduce). Instead all operations are executed in parallel. The results of an operator are then directly forwarded to following operator to be processed, which results in a pipelined execution. Flink programs written using one of the many APIs, as described in the next section, are translated internally into abstract data flow programs. These are then transformed into execution plans using logical and physical cost-based optimization. These can then be executed in the engine. The scheduler decides on the operator placement and tries to exploit data locality where possible.

Flink provides a distributed runtime environment for clusters and also a local runtime environment. Programs can, therefore, be run and debugged right in a local development environment.
environment easing development. The distributed engine adapts the execution plan to the cluster environment and, thus, can run different plans based on the environment and data distribution. Flink is compatible with a number of cluster management and storage solutions, such as Apache Tez\(^1\), Apache Kafka\(^2\) [4], Apache HDFS [5], and Apache Hadoop YARN [6].

*Stream Builder* and *Common API* translate between the runtime environment and the interfaces (API) by transforming directed graphs of logical operations into generic data stream programs that are executed in the runtime environment. The automatic optimization of data flow programs is included in this process. The integrated optimizer for example chooses the best concrete join-algorithm for each respective used case, with the user only specifying an abstract join operation.

### 3.2 Libraries and Interfaces

Apache Flink users can specify their queries in various programming languages. A Scala and a Java API are available for the analysis of data streams and batch processing, respectively. Batch data can further be processed via a Python API. All APIs offer the programmer generic operators such as Join, Cross, Map, Reduce, and Filter. In this Flink differs from Hadoop MapReduce, which only allows for complex operators to be implemented as a sequence of map and reduce phases. Furthermore, users can specify arbitrary user defined functions. Listing 1 shows a word count implementation with the Scala Stream Processing API. Analogous to this example an implementation to batch process is possible with the omission of the window specification.

```
1  case class Word(word: String, frequency: Int)
2  val lines: DataStream[String]
3     = env.fromSocketStream(...)
4  lines.flatMap(line => line.split(" "))
5    .map { (_ , 1) }
6     .keyBy(0)
7     .timeWindow(Time.of(5, TimeUnit.SECONDS))
8     .sum(1)
```

**Listing 1.** Word count implementation using Apache Flink’s Scala stream processing API.

In the first line a tuple consisting of a string and an integer is defined. Line 2 indicates a Socket Stream, which reads a text data stream line by line. In Line 4, a FlatMap-Operator is applied, which obtains lines as input, divides these by blank spaces, and converts the

\(^1\) http://tez.apache.org  
\(^2\) http://kafka.apache.org
resulting single words into the previously defined tuple format with the word as string and 1 as numeric value. Since this is a data stream query, a window is specified. This window is a sliding window with a duration of five seconds. Finally, the words are grouped and the numeric values are added up within the various groups. The print method outputs the result on the console.

In addition to its classical interfaces the FlinkML library offers a number of algorithms and data analysis pipelines for machine learning. Gelly enables graph analysis with Flink. The Table API allows for declarative specifications of queries similar to SQL. It is available as Java and Scala version. Listing 2 shows a word count implementation with the Java Table API for batch processing.

```java
1 Dataset<Word> input
2 = env.fromElements(new Word("Hello",1),
3       new Word("Bye",1),new Word("Hello",1));
4 Table table = tableEnv.fromDataSet(input)
5 .groupBy("word")
6 .select("word.count as count, word");
7 tableEnv.toDataSet(table, Word.class).print();
```

Listing 2. A word count implementation with the Java Table API for batch processing.

Initially the input is explicitly created. Line 4 first converts the Dataset to a table to then group it according to the attribute word. Just like in SQL the select command chooses the word as well as the sums of numerators. The result table is finally converted back into a Dataset and printed.

### 3.3 Storage Interaction

Streaming and batch programs load from and / or store data to persistent storage. To better understand the role of SAGE in Flink, we will describe different cases of how programs interact with storage.

To cope with the connectivity requirements of a wide range of use-cases, Flink supports a number of backends for both streaming and storage applications. The interfaces of the individual backends are abstracted into Flink connectors. Each connector is tailored to a particular backend, and handles all interaction between Flink and the backend.

As Flink supports both streaming and batch APIs, a connector is specialized to either streaming or batch processing. Therefore, the connectors can be subdivided into streaming connectors and batch connectors, respectively. Further, input and output are handled with distinct connectors. Input connectors are referred as data sources, output connectors as data sinks.

© DFKI and other partners of the SAGE consortium 2015-2018
As shown in Listing 1 and Listing 2, data sources are declared by invoking a method on an execution environment. The method specifies the type of stream or batch input, and the location of the input is given as a URL parameter to the method. In both streaming and batch processing, data are handled as tuples that flow through the system (hence: data flow). Tuples are characterized by the input type, i.e. a text file is split line-wise into a series of string tuples. Flink reads tuples "on demand", that is as fast as the system can process data. The processing rate is handled by a backpressure mechanism: if one part of the system cannot handle the current data rate, it will build up a backlog. Once the backlog surpasses a threshold, the system automatically reduces the data rate until all backlogs are within the threshold.

While Flink assumes a data source to yield a sequence of input tuples, how data is read depends on the type of storage backend and the parameters given to the connector. For example, while a text file connector sequentially reads a data file, a CSV file connector can select columns of data, therefore the physical file access is strided. More complex backends, e.g. key-value stores or SQL, can access data effectively in arbitrary patterns as given by the application's data access request. A data source can also be split into multiple ranges which Flink can read and process in parallel. This is important for exascale data processing, as reading input can be distributed over multiple Flink TaskManagers and storage nodes.

Data sinks are handled in a similar fashion to data sources. By invoking a method on an execution environment, the type and location of the output are specified. As an example, Listing 2 writes to standard output. Typically, batch data sinks are persistent storage backends and streaming data sinks are non-persistent streams (e.g. network sockets). However, this is not an inherent limitation of Flink, and both batch and streaming APIs can in principle have connectors of either type, persistent or non-persistent, as is demonstrated by the streaming HDFS data sink.
4. **SAGE Platform**

In the following we provide an introduction to the SAGE platform. The overview captures the storage architecture. Thereafter we briefly describe the main software elements, Mero and Clovis.

### 4.1 Overview

We re-iterate the top level picture of the SAGE system which is a data centric computing oriented storage system that is designed to bring compute to data locations in the context of extreme computing. The figure below is an overview of the core SAGE system, which consists of multiple storage device tiers, as shown in Error! Reference source not found. below, exploiting locally available compute at the node level (with a storage node consisting of multiple storage devices of the same type, typically). The storage tiers have more compute capability as we go up the storage system stack (with higher performing and lower capacity storage devices located higher up in the stack).

The availability of local compute in such a manner is the key point of consideration for Apache Flink to exploit.

We next very briefly describe the Advanced Object Store (Mero) and the Clovis API – as is relevant for Flink.
4.2 Mero

As we show in the picture, the SAGE system is driven by an Advanced Object Storage software platform called Mero that fundamentally manages data across the various tiers. At a very basic level Mero is an Object store, with an accompanying Key Value store. However, after carefully identifying the requirements of data-intensive extreme computing, as well as gathering the requirements from SAGE use cases (as done in Sage deliverable D1.2), many exascale-oriented components will be built on the Mero platform. More details of the individual exascale components of Mero is available from D1.2. We depict the exascale components in the picture above.

The API to assess and manipulate data within Mero is through the Clovis API which we describe next. Flink, which is a basically an ecosystem component of SAGE working on top of Clovis, will use the Mero object store through the Clovis API.

4.3 Clovis

The Clovis API consists of three separate interfaces, an access interface, an extension interface and a management interface. The access interface provides access to the Objects (Object I/O operations) and the Key Value store used to describe and index these objects. The extension interface is used to extend Mero functionalities by third parties. The management interface is used to collect telemetry information from the Mero platform for debugging and performance analysis.

The Clovis access interface needs to be leveraged by Flink for working on top of Mero.

4.4 Summary

In this deliverable we provide insights into how Apache Flink will work with Clovis. Adapting Apache Flink for Clovis and Mero provides a powerful solution and a proof point for data analytics for extreme computing and BDEC.
5. Apache Flink on SAGE

We first study the interfaces of Flink and Mero and identify the components of the data analysis framework that have to be adjusted to exploit a multi-layered storage system. Based on this analysis, we then describe three architectural designs and how they use the modified components.

5.1. Overview

As a flexible, distributed, data-flow framework, Flink is designed to suit a wide range of user groups and usage scenarios in both the stream and batch data processing paradigms. With this diversity comes the requirement to access data stored in many data formats and storage platforms. To name several, Flink currently supports HDFS, HBase, JDBC, and the always-present POSIX file systems. SAGE adds another storage platform with Mero. Mero, however, is more than just another storage platform. With the stated target of exascale storage and processing capabilities in high-performance computing applications, SAGE introduces a paradigm shift in how applications and data processing frameworks must integrate with the storage platform. In this chapter, we will describe the technical challenges Flink and Mero must solve to achieve integration of high-performance at levels. We outline several proposed design choices and explain their modes of integration. Then we will highlight the advantages and drawbacks of each design choice and weigh the best solution. Finally, we will summarize our insights and draw a conclusion.

5.2. Component Analysis

In a distributed system, the key to handling very large amounts of data is efficiently handling network communications. During a Flink computation, data is transferred over the network in bulk when loading input from a storage system to workers (known as TaskManagers in Flink), when shuffling intermediate results between workers, and when storing output data from workers to the storage system. At Flink's core is an optimizer, which job is to reduce the need to shuffle data between workers as far as possible. This leaves the load and store operations as targets for efficiency improvements. At the architectural level, these improvements come from two core innovations of SAGE: avoiding extreme data movements in the I/O stack between computation and stored data, and direct access to external data sets. Taking advantage of these innovations in Flink involves the following challenges.

Storage connection

The first challenge in integrating Flink and Mero is the basic I/O functionality. The success of the integration depends on achieving high throughput and low latency Flink-Mero communication. Bottlenecks in this layer will become prominently visible because the SAGE architecture eliminates inefficiencies throughout the storage system. Therefore, it is critical
that the storage connection does not impede performance, and must be designed with
forethought. The challenge lies in the many details and potential performance pitfalls.
Storage platforms come with APIs built for their capabilities and use-cases. To support a
large number of storage platforms, Flink abstracts platform-specific details in a storage
connector. A data source connector provides iterator-style primitives to access input tuples.
Output is similarly handed off to data sinks.

To avoid unnecessary data movement throughout the extended I/O stack, the Flink storage
connector must take advantage of Mero’s intrinsic knowledge of data locality. At a high
level, data locality can be achieved by giving Flink workers access to distinct segments of
data. This distributes and parallelizes data access over multiple workers while allowing
Mero to retain control over locality. Clovis exposes a segmented view of data objects, which
the Flink storage connector for Clovis must take into account.

As a JVM-based framework, Flink storage connectors are written in a JVM-supported
language, typically Java or Scala. Clovis, the API exposed by Mero, is written in C. Thus Flink
cannot directly interact with Clovis – a high-performance, JVM-compatible way of accessing
Mero must be found.

**In-storage computation**

The cost of moving data grows with the scale of data and the size of the system used for
processing the large amounts of data. Efficiently transferring data ultimately only delays
the inevitable: data transfers become so expensive that computations on it become
unfeasible despite sufficient computational capabilities.

Direct access to external data sets, as proposed by SAGE, drastically reduces data transfers
within the storage system and between the storage and computation systems, i.e. Mero and
Flink. As opposed to transferring data to the compute cluster, the computation is moved to
the data. Therefore, computations occur *within* the storage system, which we call in-storage
computation. The storage system either performs pre- and post-processing sub-tasks to
reduce the data that must be transferred to the computation system, or evades the data
transfer altogether and performs the complete computation without involvement of a
dedicated compute cluster. The result is higher data access performance with low latency
and high throughput, lower power consumption, and less network bandwidth consumption.
All of these metrics ultimately lower the price for processing large quantities of data.

In-storage computation is non-trivial for several reasons. In general, the computation is
subdivided into functions. Ad-hoc, each function is a black box to the executing system; the
system knows neither the intended functionality of the function, nor what it actually does.
The opaqueness of functions makes it hard for a system to reason about them. This
opaqueness is the reason why Flink specifies in its programming interface a set of
parallelization contracts (PACTs). A PACT is an interface of a second-order function which
specifies an output contract. The contracts give guarantees on the behavior of the functions
that implement them. A PACT function’s behavior can be reasoned about by the system, and thus gives the system a basis on which it can perform program transformations, optimizations, and implement fault-tolerance. However, a PACT does not specify what goes on within a function. As a function may make assumptions about the environment it is executing in (e.g. the system’s API features and runtime implementation, libraries, operating system services, etc.), PACTs are not sufficient to move an arbitrary PACT function from within Flink to another system (e.g. Mero). We call a function that makes no assumption on and does not modify it environment a pure function.

A pure function is oblivious of its environment. Hence a pure function is, in theory, trivially portable from one system to the next, while a system can further assume the function fulfills its PACT guarantees. In practice, however, at least one environment assumption cannot be eliminated: the programming language. The programming language a function is expressed in is a fundamental assumption that constrains portability of pure and non-pure functions. By implication, any system that will run a function that is pure and adheres to a PACT contract but is otherwise arbitrary must support the language runtime of the programming language the function is written in. Flink provides Java and Scala APIs, hence the Java virtual machine is the target language runtime.

The challenge for bringing Flink-compatible, in-storage computation to Mero is to find a suitable compromise between supporting system features as provided by the Flink API and runtime together with the libraries non-pure functions use as commonly found in Flink programs, and constraining the assumptions functions are allowed to make about their environment.

The ideal solution would not require changes to existing Flink programs, allow Flink programmers to retain their accustomed and established development methodologies, and at the same time accelerate the performance of Flink programs with in-storage computation capabilities of Mero.

Hierarchical storage

A hierarchy of explicitly managed storage tiers in SAGE is a novel concept, to which none of the traditional, Flink-supported storage platforms has a comparable structure or abstraction. As there has previously been no possibility or need for explicit, tiered hierarchical storage support, Flink is oblivious of a storage hierarchy, thus Flink assumes uniform storage access implicitly.

Awareness of the tiers is important when processing large quantities of data. Taking into account the expected performance characteristics of tiers can lead to improvements in application throughput and latency, as the application has knowledge of data locality in time and space. Explicit management, as opposed to a caching-based solution, lowers pressure on the storage system as data can be placed on the optimal tier from the onset.
Would the data instead be cached, it would be moved from higher to lower tiers on writes as the higher tiers reach capacity limits, and subsequently from lower to higher tiers on reads. Any unnecessary data movement between tiers costs time and energy, while consuming throughput and IOPS which might more productively be used for other data accesses.

Introducing explicit storage tier management to Flink will require changes throughout the framework. Applications should have control over which tier writes occur to. Therefore, the storage hierarchy must be exposed to applications via the Flink APIs. If an application chooses not to explicitly specify a tier, Flink must be capable of automatically deciding on the most appropriate storage tier for the particular application and use-case. This should occur at run-time, when the application’s behavior may be observed. Factors that influence this decision could be the estimated output size for batch workloads, or the current output data rate for streaming workloads. For example, the optimal tier would be the one with minimal sequential write capability faster than the output stream’s data rate. Bursts in the output data rate could be handled by temporarily migrating the stream to a faster storage tier. Furthermore, Flink could use the higher tiers of the hierarchical storage system for temporary data resulting from operator state overflows, as described below. Devising a mechanism to choose the optimal tier for general applications is an unsolved research challenge.

**Operator state spilling**

Main memory is a scarce resource in large-scale, data-intensive computations. The data-flow computation model, as implemented in Flink, avoids excessive memory usage by pipelining operators in the computation. Intermediate results between operators are typically not materialized and saved. However, materializing intermediate results of pipeline breakers, as known also in classical relational database management systems, cannot be avoided. Similarly, operator state of user-defined functions may consume large amounts of memory. In such scenarios, Flink spills operator state by temporarily writing memory to persistent storage. As operator state size depends on input data and the specific operator (which can be a user-defined function), the amount of data can grow arbitrarily large. Access speed to the data is crucial as it will determine the overall system’s throughput.

When using Mero as the underlying storage system, Flink could spill operator state to Mero as opposed to local disk. The higher reliability of the Mero storage system will contribute to better performance of Flink, as potentially time-consuming fallback state recovery of an ongoing computation is avoided at the storage layer. Computations with operator state larger than local disk capacity will become more common at exascale. In the case when multiple operators spill state on the same worker the combined space usage would be exacerbated. The use of Mero as spilling buffer would lift the storage space constraint of local disks while at the same time offering high performance at scale.
The challenges of spilling operator state to Mero are related to the challenges of adding a Flink storage connector for Mero. Both frameworks must interact efficiently for high data spilling performance. Mero’s intrinsic knowledge of data locality will play an important role to achieving throughput and IOPS. However, while Flink supports multiple storage systems via exposing an API for storage connectors, operator state spilling is embedded more deeply in the runtime system and thus will necessitate greater changes to integrate efficiently with Mero.

5.3. Architectural Designs

SAGE defines three principle modes of operation:
- Compute storage mode
- Offload mode
- Full data-centric mode

In the following, we explain the modes of operation and how from these we derive the architectural design choices.

**Compute storage mode**

![Compute Storage Mode Diagram](image)

**Figure 3 Compute Storage Mode**
In its current form when connecting to distributed storage such as HDFS, HBase, or S3, Flink and the storage solution are regarded as two distinct components. An equivalent setup in SAGE simply replaces the existing storage component with Mero. This is compute storage mode. The infrastructure hosting Flink is (possibly physically) separated from the infrastructure hosting Mero. Thus we distinguish between a compute cluster running Flink and a storage cluster running Mero, which are connected to one another via a network. When Flink accesses data held by Mero, by implication, all data is transferred in its raw form across the network from the storage cluster to the compute cluster. In compute storage mode, Mero serves as a high-performance storage system suitable for I/O-intensive applications. However, Mero is completely oblivious of the operations with which a Flink program transforms data – the program is executed within Flink on the compute cluster, barring any potential optimizations that might reduce network transfers.

Offload mode

Stepping towards a smart storage solution, offload mode integrates Flink and Mero more tightly. As in compute storage mode, Flink and Mero remain two distinct components residing on separate infrastructure. However, the barrier between the two components is weakened. While the core of a Flink program is executed within Flink on the compute cluster, selected operators are executed within Mero on the storage cluster, thereby offloading parts of the computation from the compute cluster to the storage cluster.
contrast to compute storage mode, in offload mode Mero becomes partially aware of the operations a program applies to data. Ideally, Mero executes operations that output a data transformation smaller in size than the raw data. These intermediate results are cheaper to transfer across the network from the storage to the compute cluster.

**Full data-centric mode**

In full data-centric mode we rethink where computation should occur. Both compute storage mode and offload mode execute the core program on infrastructure dedicated to computation. Access to data always incurs a transfer cost. The transfer cost can vary widely, depending on data entropy, compression methods, network technology, and details within a Flink program such as the size of intermediate results. Full data-centric mode avoids the transfer cost by eliminating the network transfer between Flink and Mero on data access. In full data-centric mode Flink and Mero reside on the same infrastructure, effectively merging the compute and storage into a single cluster.

### Comparison of Architectures

<table>
<thead>
<tr>
<th>Operation Mode</th>
<th>Current Support Level</th>
<th>Performance Improvement</th>
<th>Applications Accelerated</th>
<th>Required Changes</th>
<th>Application Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute storage</td>
<td>Native</td>
<td>Low</td>
<td>None</td>
<td>Low to Medium</td>
<td>None</td>
</tr>
<tr>
<td>Offload</td>
<td>None</td>
<td>Medium</td>
<td>Offloaded operators</td>
<td>High</td>
<td>Severe</td>
</tr>
<tr>
<td>Full data-centric</td>
<td>Partial</td>
<td>High</td>
<td>All</td>
<td>Medium</td>
<td>None</td>
</tr>
</tbody>
</table>

*Table 3: Design Comparison*

In this section we have composed and described three architectural designs to adapt Flink to SAGE’s principle modes of operation. From these descriptions we will now extract and
compare the anticipated benefits and drawbacks of our designs. A high-level overview of our findings is provided in table 3.

As a distributed data analytics platform, Flink currently operates in compute storage mode and therefore has native support for this mode of operation. Expected performance improvements will come from the high performance storage platform, but will not be able to take full advantage as applications may run into networking bottlenecks when accessing storage. The computation capabilities of the storage system are not used. Therefore, no applications are accelerated. The required changes to Flink for enabling compute offload mode primarily involve the storage connector. However, this is the minimal change that enables interaction with Mero. Applications operating in this mode could further benefit from hierarchical storage support and optimizations, and operator state spilling capabilities.

Offload mode is a novel mode of operation introduced by SAGE, thus Flink currently has no support for this mode. It builds on compute storage mode and provides the same performance improvements from the high performance storage system. In addition, Flink will have the capability to offload a narrow range of operators to the storage system, improving performance from applications, which make extensive use of these specific operators. Applications that do not use these operators will not see performance improvements beyond what compute storage mode offers. The estimated required changes are high, as numerous components within Flink will have to be adapted for offloading computations to an external system, and there will likely be intrusive restrictions to user-defined functions.

Full data-centric mode is also a novel mode of operation introduced by SAGE. While full data-centric mode shares several components with compute storage mode, its architecture is fundamentally different. As full data-centric mode moves the entire computation to the storage cluster, the Flink framework will move with the computation. The expected required changes to Flink will be higher than for compute storage mode because Flink will be more tightly integrated with Mero, but lower than offload mode because functions will presumably remain within Flink. For the same reason, applications will not see functional restrictions related to acceleration. As all computation takes place on the storage cluster, all applications will benefit from accelerated storage access. The primary beneficiaries will be applications bound by the storage system. Potentially, compute storage mode would suite CPU-intensive applications better. However, also for storage intensive applications the balance between processor and storage performance is important as not to bottleneck applications due to lack of processor resources. For further information on the balanced design of cluster hardware we refer to WP1.

Each of the architecture designs comes with potential implementation pitfalls. In this high-level architectural design document we do not cover low-level implementation issues. We will detail the issues of an actual implementation in later deliverables.
## 5.4. Requirements

<table>
<thead>
<tr>
<th>Priority</th>
<th>Requirement</th>
<th>Created / Modified Component</th>
</tr>
</thead>
</table>
| 1        | High-performance storage connection | Mero storage connector in Flink  
|          |                                 | Java API in Mero                                                                            |
| 2        | In-storage computation (offload mode) | Pure PACT functions in Flink  
|          |                                 | Modified public API in Flink  
|          |                                 | Modified scheduler in Flink  
|          |                                 | Modified storage connector in Flink  
|          |                                 | In-storage computation framework in Mero                                                     |
| 2        | In-storage computation (full data-centric mode) | Resource manager in Flink / Mero |
| 3        | Multi-tiered storage             | Modified public API in Flink  
|          |                                 | Modified runtime in Flink  
|          |                                 | Modified scheduler in Flink  
|          |                                 | Modified storage connector in Flink |
| 4        | Operator state spilling          | Modified runtime in Flink  
|          |                                 | Modified storage connector in Flink  
|          |                                 | Java API in Mero |

Table 4: Apache Flink Requirements

From the component analysis and architectural designs we deduce a list of requirements for integrating the Apache Flink data analytics platform with SAGE. For each requirement we include a list of components that will have to be either created or modified to satisfy the requirement. The findings are summarized in Table 4.

The first priority is to let applications interact with data stored in Mero. This requirement applies to all architecture designs. We must create a storage connector for Mero that is able to act as data source and data sink for batch processing applications. We will consider a streaming data sink should applications require streaming capabilities (our current applications do not). As Clovis currently interfaces only with native applications, Mero will have to be modified to support applications running in a JVM.

The second priority is split into two parts because there are two architectural designs that satisfy the requirement. This requirement allows applications to move closer to storage and reap benefits of better storage access. We will first consider the offload mode design.

In offload mode, Flink applications will be given the option of defining pure functions which are offloadable to the storage framework. Function contracts (PACTs) for defining
offloadable functions will be marked in the public API. To perform offloading, the scheduler will be made aware of offloadable PACTs. As such functions are limited to the beginning and end of an execution pipeline and therefore close to storage, the offloading will occur within the storage connector. The interface used for offloading will take place within Mero. In full data-centric mode, comparatively less changes will be needed as Flink’s native function shipping framework will be used. The changes are concentrated to deploying Flink alongside Mero, and managing the interactions that take place between the two frameworks (resource configuration, competition, conflicts, etc.). These will be handled by a new resource manager component that will be embedded in either Flink or Mero.

As third priority we have identified the requirement for supporting a multi-tiered, hierarchical storage framework and applies to all designs. This will allow applications to optimally use the characteristics of the storage tiers to gain advantages in throughput, access latencies, resiliency, durability, and further metrics. Applications will be able to manage tiers explicitly in Flink’s API, as metrics such as resiliency depend on the use-case, and writers of high-performance software may wish to hand-tune their applications. The Flink runtime and storage connector will have to be modified accordingly. We will also consider modifying the scheduler to automatically select the best tier based on an application’s behavior.

Finally, the fourth priority is to support scaling operators beyond memory capacity. The requirement is spilling operator state. This requirement applies to all designs. As Flink supports spilling operator state to disk using POSIX, we must extend this functionality to Mero. Modifications involve the runtime system and the storage connector. As in the first priority, Mero support for JVM programs will be necessary.
6. Conclusions and Next Steps

This deliverable analyzes the Flink and Mero frameworks, and based on the analysis we identified the requirements in Flink in order to support exascale data analysis applications. These include a high-performance storage connection to Mero, in-storage computation, multi-tiered storage, and operator spilling. We have described the necessary changes to Flink for complete integration with the Mero hierarchical storage framework, namely the storage connector, runtime system, scheduler, public API, resource manager, and Mero API, and examined the technical challenges these changes will have. Furthermore, we have described architectural designs based on the SAGE operation modes that integrate Flink with the SAGE platform. For each of the architectures we have listed the characteristics, and compared them with each other in depth.

The next step in WP4 is to specify the run-time system of Flink and document the requirements for the underlying software and hardware layers. In parallel to the specification, we will begin prototyping the run-time system integration, which will give us more insights into the system design that will support our specification.
References


