# Cherry: A Distributed Task-Aware Shuffle Service for Serverless Analytics

Nikolaos Nikitas
*CSLAB NTUA*
Athens, Greece
nnikitas@cslab.ece.ntua.gr

Ioannis Konstantinou
*University of Thessaly*
Greece
ikons@uth.gr

Vana Kalogeraki
*Athens University of Economics and Business*
Athens, Greece
vana@aueb.gr

Nectarios Koziris
*NTUA*
Athens, Greece
nkoziris@cslab.ece.ntua.gr

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671899

***Abstract*—While there has been a lot of effort in recent years in optimising Big Data systems like Apache Spark and Hadoop, the all-to-all transfer of data between a MapReduce computation step, i.e., the shuffle data mechanism between cluster nodes remains always a serious bottleneck. In this work, we present Cherry, an open-source distributed task-aware Caching sHuffle sErvice for seRveRless analYtics. Our thorough experiments on a cloud testbed using realistic and synthetic workloads showcase that Cherry can achieve an almost 23% to 39% reduction in completion of the reduce stage with small shuffle block sizes, a 10% reduction in execution time on real workloads, while it can efficiently handle Spark execution failures with a constant task time re-computation overhead compared to existing approaches.**

***Index Terms*—Big Data Analytics Frameworks, Distributed Systems, Cloud Computing, Serverless Architecture**

## I. Introduction

Large scale data processing and analytics frameworks, either as on-premises or as fully managed solutions like Apache Spark [1], Hadoop MapReduce [2] Flink [3], Presto [4] or Google's Cloud DataFlow [5], have emerged as the de facto standard for companies in the last decade, since vast sizes of data are generated and required to be processed. These frameworks are utilized in data-intensive circumstances, offering a performant, distributed and scalable solution that can be seamlessly adapted in cloud-based environments.

The aforementioned systems' architecture is heavily influenced by the MapReduce paradigm: they all offer a fully-distributed data parallel computation engine where the entire execution is split into a pipeline of multiple stages and each stage is split into small computation units (i.e., "tasks") which are assigned to data "chunks" by a centralized scheduler. In the case where intermediate data either between stages or between tasks of the same stage need to be stored and exchanged, all frameworks employ an internal data addressing and exchange mechanism that deals with data partitioning, shuffling and/or sorting [6]–[10] to handle intermediate task state. The efficient intermediate state handling is of paramount importance for the entire system performance and all engines try to address it in an optimal manner.

One way to deal with intermediate state (i.e., shuffle data) is to dissagregate it from the compute infrastructure by introducing an external "Shuffle Service" that offers the intermediate state management through a separate infrastructure [11]–[17]. This approach has a dual benefit: on one hand it entirely removes state from the computation pipeline rendering it completely "stateless", thus allowing a de-facto stateful workload to be seamlessly executed in a serverless manner, and on the other hand it allows for targeted optimizations both at state management infrastructure and system implementation.

One of the most prominent distributed processing frameworks, namely Apache Spark, which bases its computation in the notion of "RDDs" also follows the generic MapReduce computation pattern. A Spark cluster currently executes workloads by initiating an External Shuffle Service (ESS) [6] on each worker node, that is required to run continuously and constantly maintain state. The ESS functions as a proxy through which both node-local and node-remote executors can fetch the needed intermediate shuffle blocks when they are assigned a reduce task at a reduce stage. The main problem is that if a specific node crashes, all the intermediate data stored are lost and a part of the job lineage has to be re-executed. Even a whole node deallocation is prohibitive due to upcoming data loss. Another noteworthy drawback is that ESS is not well suited to containerized environments, like Kubernetes [18] or YARN [19], where processes are required to be isolated and stateless, since, although there are some solutions, there is no serverless behavior and containers need to be constantly up.

Furthermore, in a shuffle read phase where reduce tasks try to fetch intermediate data that are not locally located, the disk I/O procedure can be a major obstacle and can delay Spark workloads significantly due to the limited read/write throughput that HDDs set. More specifically, this constraint is of great importance when Spark executors are reading shuffle blocks of small sizes and IOPS are restricted, causing the significant degradation of workload execution time.

In this work we present Cherry[1], a distributed shuffle service that offers the following primary features:

- The introduction of a remote disaggregated storage engine that stores intermediate shuffle data between stages, making the execution of large-scale analytics workloads completely stateless. This mechanism provides fault tolerance since there is no data lost upon node or worker

[1]code available at https://github.com/nikoshet/spark-cherry-shuffle-service

failures and state is disaggregated, offering seamless scaling of operations and making the execution of Spark workers ephemeral.

- An implementation of a look-ahead caching policy on the remote storage side that is task-aware and aims at improving shuffle blocks fetch time, avoiding I/O bottlenecks. The suggested caching policy that Cherry provides has a significant performance improvement on workload executions where enormous sizes of shuffle data are required to be fetched by reduce workers.
- Cherry follows a modular architecture to facilitate easy framework integration with existing systems while it is currently seamlessly integrated with Apache Spark in a pluggable way, enhances its capabilities, and requires zero code changes and modifications to the existing Spark workloads. Moreover, it utilizes Kubernetes as a container orchestration manager, for automating the management, deployment and scaling of containerized Spark components.

## II. Background and Motivation

The aim of this section is to give motivation and present the background for Cherry. Subsection II-A discusses the procedure of intermediate shuffle files in Spark. Subsections II-B, II-C and II-D present the main problems that can be found in the existing implementation of Spark when executing big data workloads.

### A. Existing Approach in Spark Shuffle Data Management

The shuffle operation where intermediate data are being temporarily stored and later fetched exists in several map-reduce like frameworks that are used in data analytics workloads. Several services that offer this functionality have been implemented [3], [6]–[10]. We selected Apache Spark to integrate Cherry with, although as we discuss in Section VII we argue that Cherry can be easily integrated with other MapReduce-inspired frameworks, since they follow the same architectural principles which we consider in our modular design.

In the Spark computation engine, the DAGScheduler, which is located in the Spark Driver (the user program's entry to the Spark execution "world", see right side of Figure 1), is responsible for creating a DAG (Directed Acyclic Graph) of the computation, and splits a job in stages based on where the intermediate data partitioning (i.e., shuffle) operation is required to take place among the Spark executors over the cluster network, and each stage in separate tasks, according to the available job parallelization. The workers take on a task that has been assigned to them, and initiate the executors that will carry the necessary computations. The data are divided in partitions accordingly.

In a Spark DAG, there are two types of data dependencies that describe the tasks, namely narrow and wide (Section 4 of [1]). The partition of the child RDD that is assigned on a task and is dependent on at most one parent RDD, consists a narrow dependency. Common transformations are map and

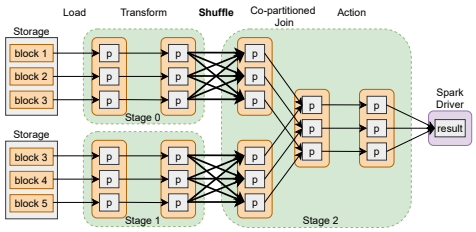


Fig. 1. DAG: A visual representation of RDDs and the operations being appplied on them for a Spark job.

filter, there is no data shuffling required over the network, and pipelining is feasible. Accordingly, when the partition of the parent RDD is relied on by many child RDDs, this creates a wide dependency. Typical transformations are reduceByKey and join. The tasks that require to read intermediate shuffle data are named reduce tasks, while the tasks that create data or read existing data as input are map tasks.

On wide dependencies, the shuffle data (i.e., intermediate data) need to be transferred around different nodes, a common phase in the MapReduce paradigm. The shuffle data that are generated after each map task, are block files that include sorted data per reduce partition, as well as index files that include offsets and lengths of these blocks. The map executor, after processing its respective partition of an RDD that is responsible for, creates the shuffle block data and index files in an explicit format. Reduce tasks, firstly, are required to remotely fetch their respective shuffle data, if the latter are not available locally, in order to proceed with an assigned computation. In Figure 1 we illustrate a DAG of a Spark job from the procedure of data loading from the storage engine until the final result is sent to the Spark Driver.

Spark has two solutions for dealing with intermediate data. Firstly, Spark executors that are assigned a reduce task get informed from the Driver about the nodes where the required shuffle data are stored. Thereafter, they can directly fetch the data from the executors that run on these nodes and serve those files. Another solution that Spark offers is the use of the ESS. A Spark cluster can execute a workload by initiating an ESS on each Worker node. It is vital for the ESS to run continuously and constantly maintain intermediate state. Both node-local and node-remote executors can retrieve the respective shuffle data when they are assigned a reduce task by the Spark Driver. The shuffle operation is a procedure that occurs widely in Spark analytics jobs that process TBs or PBs of data, and thus, we believe that it is worth improving its performance.

### B. Disk I/O Performance on Intermediate Data

The default External Shuffle Service of Spark fetches blocks from disks, a process that is subject to great I/O overhead and limited read/write throughput of HDDs because of random file seeks. Each shuffle file that is created after a map task, consists of a certain number of blocks, i.e., partitions. This number is specified by the number of tasks on the next stage of the Spark job. In addition, each block of data will be required only one time for a specific reduce task, but a shuffle file

will be requested by all reduce tasks. To put it differently, a wide-dependency stage with M tasks that requires shuffling will create M shuffle files on the map side, and if N tasks exist on the reduce side, $MxN$ connections will be created due to $MxN$ fetched shuffle blocks.

The aforementioned challenge that occurs in Spark workloads takes place especially when the shuffle intermediate data that are required to be fetched over the network are small in size, and thus, the I/O bottleneck occurs and degrades the performance of Spark applications in large scale. At LinkedIn, the average shuffle block size of its production Spark clusters is only a few KBs [11]. A solution would be to use less tasks per stage, and hence, bigger shuffle block sizes. However, this case does not depict the real cloud environments where the clusters consist of hundreds of separate nodes and, thus, making the use of small shuffle file sizes inevitable. More specifically, the size of shuffle blocks depends on the data set and cluster size, and can end up small throughout a complex long-running workload. When it comes to the selection of the storage type that will hold the occurring shuffle data of any Spark workload execution, although SSDs provide better I/O performance than HDDs, they are more expensive. For example, AWS prices a GB of SSD 4 to 5 times higher than a GB of HDD per hour [20].

### C. Fault Tolerance of Spark Workloads

Another noteworthy downside occurs with both vanilla Spark and Spark with ESS. More specifically, Spark tries to achieve fault tolerance by persisting shuffle files on disk instead of memory in case a Spark executor fails, and requiring a shuffle service to be running continuously. Nevertheless, this mechanism does not include the possibility of a crash of a specific node from the Spark cluster, since all the intermediate data that were stored at the latter will be lost. As a consequence, a major part of a lineage has to be inevitably re-executed, and a shuffle re-computation is really expensive. In cloud environments, deallocation of whole nodes or Virtual Machines within a cluster are a common phenomenon for maintenance or upgrade of hardware. However, this scenario is prohibitive for the current Spark implementation due to upcoming data loss. Additionally, the enforcement of continuous uptime of the shuffle services has the impact of the unnecessary allocation of compute resources even when there are no executors running in a specific node.

### D. Isolation and Serverless Execution

Finally, another problem that needs to be highlighted is that the whole Spark architecture, and more specifically ESS, is not adapted to containerized environments, like Kubernetes or YARN, where processes are required to be isolated and stateless. Furthermore, the Spark Workers that run on the same node have access to its other's shuffle service, violating any isolation prerequisites or policies between components that may be a mandatory feature in a cloud cluster. In addition, there is a wide interest in disaggregated cluster deployments in the cloud, where the compute engine is located remotely from storage engine, and workloads can run in a serverless manner seamlessly, without having any intermediate state. The same requirement is for containerized environments, where containers should be started or terminated without the danger of losing any state or intermediate data. The current implementation of Spark with Kubernetes does not complete this requirement, since the ESS of each Spark Worker that is executed in a single container stores any intermediate state and needs to be constantly up. Consequently, it is a single point of failure. Thus, there is no fault tolerance and serverless behavior in existing available solutions.

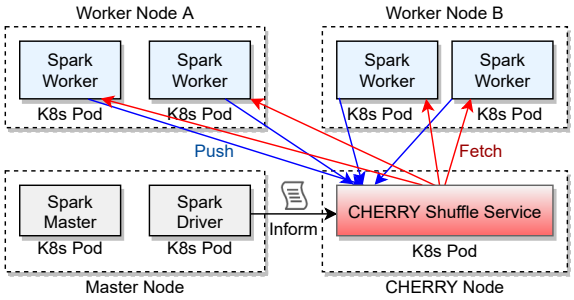


Fig. 2. *Cherry* Topology.

## III. System Architecture Overview

In this Section we present Cherry, a distributed shuffle service for Spark, that is implemented with the aim of addressing the aforementioned challenges and, thus, improve the execution time of Spark analytics workloads in large scale. Cherry is an open-source project that is built on top of Apache Spark and its main topology is illustrated in Figure 2. Although our suggested service for the shuffle operation is Spark-specific, we believe that this architecture and mechanisms can be ported on similar systems, as we analyze in Section VII.

Our proposed shuffle service can be integrated seamlessly with Spark, without requiring any modifications to existing Spark workloads (i.e., existing client code). Cherry uses Kubernetes as a cluster management tool. Kubernetes offers great flexibility, as it can easily allocate and deallocate nodes from the Kubernetes cluster, deploy containers and receive heartbeats, as well as achieve isolation between the Spark components. Each module of our system is deployed in a separate Kubernetes (i.e., K8s) Pod to achieve isolation. More specifically, we decided to deploy the Cherry Pods in separate nodes in order to store any intermediate shuffle data and state of the Spark workloads in a disaggregated manner. Furthermore, there are specific nodes where the Spark Worker Pods with one executor each are running, based on user configurations and available resources. On a separate node, the Spark Master and Spark Driver pods are deployed.

Another component that is implemented and is running on the Master node is the Metadata Service. Its main role is to keep track of the alive Cherry Pods that are part of the Spark cluster. More specifically, when the Cherry shuffle service Pods are initiated, they firstly get registered on the

Metadata Service, and the latter stores their network information. In addition, when the Spark executors are initiated by the Spark Worker Pods, they request from the Metadata Service the list of the aforementioned Pods in order to retrieve the available locations to push and fetch their respective produced intermediate blocks. Finally, the Cherry Pods send frequently heartbeats to the Metadata Service, since the latter needs to keep track of these and know if a failure occurs. The intercommunication between the Spark components is implemented through the RPC communication protocol.

## IV. Implementation

This section analyzes in great detail the core mechanisms of Cherry. We illustrate CHERRY's detailed orchestration and data movement processes in Figure 3, with a more specific description subsequently. With light blue we depict the existing pipeline of Spark. With light red (steps 2, 4, 5) we represent the additions that we implemented so as to integrate Cherry into Spark.

- In step 2, the Spark executor, after completing its assigned task computations, pushes the shuffle data and shuffle index files to one of the available Cherry Pods based on a round-robin pattern, and the latter stores them in its local disk.
- In step 4, the Spark Driver sends the required details of the upcoming task launches (i.e., blocks that will be requested by the next reduce computation "wave") to the Cherry shuffle services so as to decouple it.
- In step 5, each Cherry Pod processes the acquired knowledge so as to proactively bring the shuffle data that will be requested from each upcoming task.

### A. Pushing Shuffle Files

We decided to follow a disaggregated approach to implement the distributed Cherry shuffle services, so as to maintain the intermediate data and state of any running Spark workloads away from the Spark workers. This feature transforms the Spark workloads into fault tolerant executions, since, if a node where workers are running crashes, no shuffle data will be lost, and hence, any re-computation of the DAG will be avoided. Furthermore, this mechanism of maintaining a shuffle storage remotely allows the Spark Worker Pods to run seamlessly in a stateless manner. Furthermore, the flexibility that Kubernetes offers in deploying, managing and removing these Spark components with preconfigured allocated resourses, as well as increasing and decreasing the number of these instances that are being executed at any point in time, makes the Spark framework a completely serverless analytics engine.

The Spark Worker Pods obtain the list of available Cherry shuffle services at the initiation of a Spark job from the Metadata Service, since the Cherry Pods are already registered to that. In order for the latter to end up having an approximately equal amount of intermediate data, the Spark executors push their shuffle files in a round-robin pattern to the Cherry Pods. When an executor is initialized from the Spark worker, it registers with all the available Cherry shuffle services in the cluster, in a similar manner as it takes place with the ESS of Spark. The details acquired by the Cherry Pods from the executors are their IP and Port, as well as their local directories that the latter will use to store their shuffle files.

When a map stage initiates, the Spark Driver schedules the map tasks on selected executors (step 1, Figure 3), based on the available resources of the Spark Workers. When an executor completes the computation of the map task, it produces a shuffle data file alongside a shuffle index file that stores locally, and then pushes them to the Cherry Pod that has been defined by the round-robin mechanism via particular RPC messages (step 2, Figure 3). When this process terminates, the mapper notifies the Driver about the completion of its task (step 3, Figure 3). Cherry persists the shuffle files locally by cloning the file path names of the executors, which are known from when the registration of the latter took place. This technique makes straightforward the retrieval of the requested blocks to be fetched later on by reducer executors. We observed through evaluation and measurements that the time that is required to push shuffle data from mappers to the Cherry services does not add any supplementary overhead.

The MapStatus component of Spark has a crucial role for matching the created shuffle files of map tasks with their location, since the reducers will need to consume this information and trace their location. More specifically, it is the result that is returned by a map executor to the DAGScheduler of the Spark Driver after its task completion, and includes the address of the Cherry service that the shuffle files are maintained alongside the sizes of outputs per reducer. When a reducer executor is initiated subsequently, it will receive the updated MapStatus instance of the location of the shuffle files that are required for its assigned execution by the Spark Driver. In essence, the MapStatus is a data structure that holds information about the location of the intermediate shuffle files, where "map" tasks write to and "reduce" tasks read from.

### B. Look-Ahead Task-Aware Caching Policy

In Spark's architecture, the all-to-all communication and message exchange over the cluster network can not be avoided. Furthermore, a huge number of random file seeks is required in the shuffle operation between stages when small sizes of intermediate shuffle data are fetched by the reducer executors in shuffle heavy workloads. This phenomenon has a major impact on the system performance, since the latter greatly degrades due to the limited random I/O throughput of HDDs that is a severe obstacle and causes a major slowdown. Thus, concerning the challenge of the disk I/O degradation of the performance of the system on these workloads, we decided to create a policy that is task-aware.

**Caching shuffle blocks instead of shuffle files.** We firstly opted to cache in Cherry's memory the whole shuffle file every time that it would be requested. However, out-of-memory errors would occur, since, on large-scale workloads with TBs or PBs of data each shuffle file is large in size and we could not proactively cache many intermediate files. In addition, each block would be required as many times as the partitions that

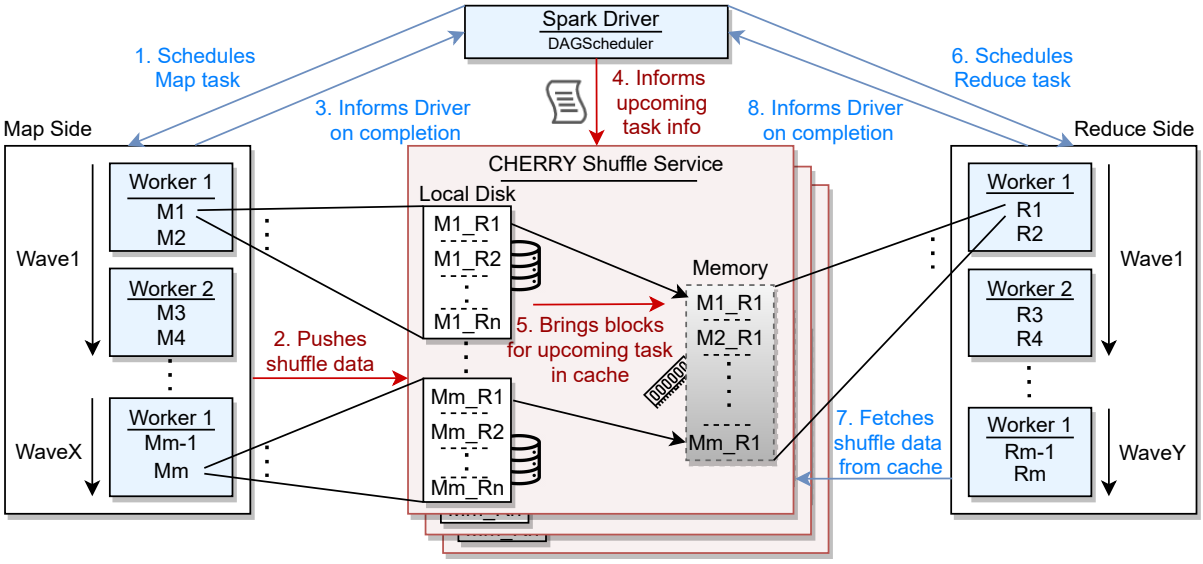


Fig. 3. *Cherry's* orchestration and data movement.

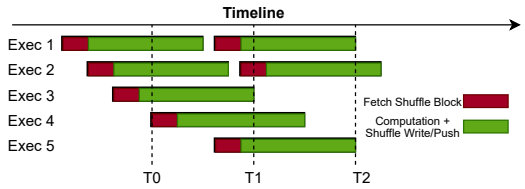


Fig. 4. Reduce stage of a Spark job. At each moment Cherry aggressively caches the shuffle blocks of the upcoming tasks.

it contains for each reducer. Consequently, many additions and removals of the same shuffle data file in cache would occur causing a lot of random evictions due to restricted available memory resources and, thus, adding overhead to the performance of Cherry. Thus, we ended up aggressively caching only the required blocks of the shuffle data files per Spark task, according to the MapStatus component.

In Figure 4, a typical reduce stage of a Spark job that consists of 7 tasks is illustrated. Using this as an example, we outline Cherry's task-aware caching policy. At the moment $T_0$, Cherry will have already acquired and processed the appropriate information from the Spark Driver about the first task of the executor 4, and will have cached the blocks that the latter will require to fetch. Cherry will evict each block that will successfully get fetched. At the moment $T_1$, Cherry will be serving cached shuffle blocks to the executor 2 for its second task . Finally, at $T_2$, Cherry will only have in memory the blocks that were not fetched because they were located locally in the executors. Note that at any specific moment in time (vertical lines in Figure 4), only the blocks of the active reducers that are currently fetching data (i.e., red rectangles) will be stored at Cherry, thus limiting the maximum amount of required cache memory irrespective of the entire dataset size.

The DAGScheduler process of the Spark Driver is responsible for managing and orchestrating the task assignment to Spark executors and broadcasts all the appropriate details to them. The Spark executors of a cluster take on a task in 'waves' accomplishing the maximum parallelization possible, if there is no data skew. Consequently, the adjustment that was implemented on the DAGScheduler is explained subsequently. At the phase of the reduce task creation, the Spark Driver predefines the exact task execution order. At this point Cherry consumes this knowledge (step 4, Figure 3), and can then easily discover the exact block IDs that will be requested beforehand (i.e, look-ahead). By acquiring this specific data, the latter is capable of aggressively caching only the needed data partitions of each shuffle file on task-level by buffering them in memory (step 5, Figure 3), and have them ready to be fetched by the reducers.

**Implementing the caching policy.** A reduce task is initiated by the Spark Driver (step 6, Figure 3) and proceeds on remotely fetching the necessary shuffle partitions before starting any computation (step 7, Figure 3). For each cached shuffle block that is fetched, Cherry can discard it, since it will not be requested again, in order to avoid cache overflow. Then, it can simply fill the memory with the next upcoming blocks according to expected tasks required to be executed. If the available memory resources get limited, FIFO eviction of shuffle blocks takes place, if necessary. The executors that complete their reduce tasks, notify the Spark Driver that their task is completed (step 8, Figure 3).

In order to find and proactively store in cache the shuffle partitions of each upcoming task we need to acquire specific information from the Spark Driver. The Spark Driver is responsible for bookkeeping the information of mapping each map index to its respective location in a MapStatus array for each stage. The MapStatus array of a parent stage is complete after the latter is finished. At this moment, it will include the map output locations of shuffle blocks in an efficient byte format ready to be sent to the reduce tasks of the child stage. Each Spark Worker acquires once per stage the MapStatus array of the shuffle that the running stage is dependent on. Additionally, each executor that is assigned a task receives a

message that contains the description of the assigned task to decompose and process.

After the DAGScheduler computes the DAG of stages for a job and the sequence of the tasks, it submits them to the TaskScheduler. Subsequently, we collect the shuffle Ids that this stage is dependent on and its stage Id, and we serialize the MapStatus arrays for its parent stages. Then, we send an RPC message asynchronously to the endpoint of each Cherry Pod with this information. The Spark Driver, then, continues to task assignments on executors. Before each and every launch of a new task in a Spark job, it sends some information relative to this task to all available Cherry endpoints. Each Cherry continues on processing this information and, consequently, on caching the upcoming shuffle blocks that will be requested shortly for faster access.

This procedure is described more thoroughly in Algorithm 1. The functionality of finding the mappings of each shuffle block ID is similar to vanilla Spark with small changes. Thus, Cherry processes the data and returns an iterator of the shuffle block IDs and corresponding shuffle block sizes requested. Each Cherry Pod stores an ID for every shuffle block that it receives from the Spark executors, that includes the Cherry's IP and Port and the executor Id that pushed each block. Hence, we then filter the iterator to keep only the requested blocks that are stored locally on each Cherry. Later, the filtered iterator of blocks is added in a queue. From this queue we iteratively fetch the objects and discover each shuffle index and data file. Its length and offset are retrieved from its respective shuffle index file. Finally, we cache each shuffle block.

The data structure that we proactively store the shuffle blocks is a LinkedHashMap, since it offers selection of number of buckets and FIFO eviction by default. Each key is a unique string that includes the path of the shuffle data file in the local disk, the length and the offset requested. There will be no key collisions since each partition will be requested only once. The messages that are exchanged when the procedure of fetching shuffle block data takes place are the same with vanilla Spark. Additionally, if a shuffle block has not been cached in time when requested, it is served with the way that native Spark works. The shuffle blocks that were cached but not fetched because they were located locally in the executors that would require them, get evicted at the end of the Spark workload, so as to free up memory resources for new jobs. Also, if a reducer crashes after its respective shuffle block has been evicted from Cherry's cache, it simply gets re-cached in order to be fetched from the new reducer, so the execution continues normally.

**Algorithm 1:** Caching Shuffle Blocks

***taskDesc***: description info for each task that is launched
***stageId***: stage Id of the launching task
***mapStatuses***: array of MapStatus instances that keeps mappings from map index to output locations for each partition of a stage
***shuffleId***: Id of the shuffle that a task depends on
***blocksIter***: iterator of blocks to be fetched for a task
***q***: queue to store blocks before processing their data
***cache***: cache to store shuffle blocks

```
1  begin
2      blocksIter = ConvertMapStatuses
         (shuffleId, stageId, mapStatuses, taskDesc)
3      blocksIter = Filter(blocksIter)
4      q ← {}
5      for all block in blocksIter do
6          q ← q ∪ block
7      for i ← 1 ... len(q) do
8          blockData = GetBlockInfo(q(block_i))
9          blockKey = GetUniqueKey(blockKey)
10         blockValue = ManagedBuffer(blockData)
11         cache.put(blockKey, blockValue)
12 end
```

### C. Efficient Memory Usage and Low Cache Miss Rate.

With Cherry, the memory resources required are low, since only a portion of tasks from a stage is being executed at each moment in time, since the available executors in a Spark cluster are usually less than the number of these tasks. We wanted to benchmark if there are many cache misses when trying to locate shuffle blocks, in order to serve the latter to executors, based on the size of available memory. Our benchmark included an execution of a shuffle synthetic workload of 40GB with 10KB shuffle block size, 10 executors and 10 Cherry Pods. We also examined using different percentages of total available cache relative to the shuffle data volume. Through our experiments we concluded that the cache misses are below 1% for all test cases, even if all shuffle data could presumably fit in memory. This happens because only a part of the shuffle blocks required is being fetched at a time, and Cherry optimally leverages any acquired information from the Spark Driver for a few upcoming reduce tasks by caching only these shuffle blocks and maintaining its memory footprint low. Thus, Cherry does not require a lot of memory resources to perform greatly.

### D. Fault Tolerance.

In large-scale systems in the cloud the node crashes is a pretty common phenomenon. Accordingly, we believe that in a Spark job there is a high chance of loss of a Spark Worker along with its local state. Cherry's mechanism of pushing and storing remotely any ephemeral shuffle data from executors provides fault tolerance in Spark workloads. More specifically, Cherry leverages this feature and allows Spark to be executed as a serverless framework, since in case of Worker node failures or deallocations, no shuffle state will be lost.

Additionally, our implementation offers resilience even in case of failures of the Cherry Pods. More specifically, the Metadata Service is responsible for keeping track of the alive Cherry services in a cluster. In case it stops acquiring heartbeat messages from a Cherry Pod, it immediately informs the Spark

Workers. Thus, on upcoming map tasks the executors push their shuffle intermediate data to the rest of the available Cherry Pods. Also, on an upcoming reduce task, we keep both the Cherry location as well as the map executor location in case of native Spark operation of each shuffle partition in order to facilitate every upcoming reduce executor. The reducers will (i) firstly examine if the respective Cherry Pod is alive, and if not, they will (ii) fetch the intermediate data directly from the respective Spark Worker, i.e, Cherry is being bypassed and Spark defaults to the native way of shuffle data access. This is possible since the Spark Workers temporarily keep their produced shuffle blocks.

## V. Evaluation

In this section, we present the results of our evaluations for Cherry. Through different types of benchmarks we showcase that Cherry can reduce the I/O latencies by proactively caching the shuffle blocks on task-level and, thus, improve the performance of Spark workloads, while offering fault tolerance via its serverless architecture and having low resource footprint.

### A. Evaluation Setup

We have evaluated Cherry on synthetic and realistic benchmarks. Our evaluation setup consists of 11 physical nodes, each of which has 1 Quad-Core E5405 Intel Xeon® CPU @ 2.00GHz and 8GB RAM, and are connected with a 10 Gbps Ethernet link. Each Spark Worker Pod allocates 1 CPU core and 2GB RAM and run one executor, and are completely stateless since only soft state is stored temporarily and shuffle intermediate data are maintained remotely with from the Cherry Pods. Each Cherry Pod has 1 CPU core and 2GB RAM available. The Cherry and Spark Worker pods utilize HDDs as a permanent storage. Additionally, the Spark Master, Spark Driver and Metadata Service Pods allocate 1 CPU core and 1GB RAM each. One physical node is dedicated for the execution of the Spark Master, Spark Driver and Metadata Service Pods, while five are dedicated of the Spark Workers and five for the Cherry shuffle services. We believe that a fair comparison of Spark with our implementation is the same number of External Shuffle Services versus Cherry shuffle services. Thus, we decided to run 10 Spark Worker Pods and 10 Cherry Pods for all our experiments. Additionally, since both Spark with its ESS as well as vanilla Spark perform similarly, we will mainly compare our implementation with Spark with its ESS, and use Vanilla Spark only for our synthetic shuffle workload.

### B. Synthetic Workload Evaluations

We created a synthetic workload and emphasize on the shuffle operation of Spark. The benchmark consists of a map and a reduce stage of a shuffle heavy synthetic workload similar to [11]. With this benchmark we can compare the performance of CHERRY against Spark using ESS and evaluate them on the reduce stage time, fault tolerance, scalability and resource consumption.

TABLE I
EXPERIMENT CONFIGURATIONS. EACH ROW SHOWS THE TOTAL SIZE OF THE SYNTHETIC DATASET, THE NUMBER OF MAP OR REDUCE TASKS, AND THE SIZE OF EACH BLOCK.

| | Size | # M/ # R tasks | Block size |
|---|---|---|---|
| 1 | 50 GB | 400 | 312.5 KB |
| 2 | 50 GB | 2000 | 12.5 KB |
| 3 | 50 GB | 4000 | 3.125 KB |

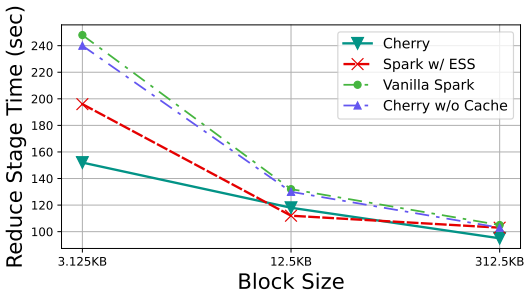


Fig. 5. Experiment results of our benchmark with synthetic workload. Smaller block sizes affect severely the I/O performance of Spark with ESS. On the other hand, Cherry achieves a better overall performance on the completion of the reduce stage.

*1) Completion Time:* The optimized performance of Cherry can be depicted on the shuffle phase where reduce tasks require to fetch intermediate shuffle data. Therefore, we monitored the total reduce stage time of our Spark workload. The map stage completion time is the same for all options since there is no optimization there and the time spent on the process of pushing shuffle data on the Cherry Pods is insignificant. We decided to test our benchmark on different number of tasks per stage and block size, and used the same number of map and reduce tasks on each stage. The number of map tasks equivalents to the number of shuffle files created, while the number of reduce tasks is the same with the number of blocks per shuffle file. Table I shows the experiment configurations.

Figure 5 shows our evaluation results when we compare our optimized shuffle service against Spark with ESS, Vanilla Spark, and Cherry without caching. On small block sizes of 3.125KB, the Disk I/O bottleneck is obvious and degrades the performance of the workload execution with Spark. On the contrary, Cherry manages to overcome this problem by utilizing its look-ahead caching policy on each task and have its requested shuffle block in memory before the fetching operation, since any I/O will take place beforehand and not in the critical moment of shuffle block fetching. Cherry without caching has a similar performance as Vanilla Spark, but can leverage its features on analytics workloads, such as fault tolerance and disaggregation.

Cherry achieves an almost 23% reduction in completion of the reduce stage time with 3.125KB shuffle block size against Spark with ESS, and an almost 39% reduction against Vanilla Spark. More specifically, Spark with ESS needs 196 seconds for the completion of the reduce stage that includes reading the intermediate data with a 3.125KB shuffle block size, Vanilla Spark requires 248 seconds, while our system

needs only 152 seconds. As the block size increases for the other 2 experiments, the performance of the 4 implementations improve similarly. Thus, Cherry offers a great gain in the reduce phase when it comes to reading small shuffle block sizes.

*2) Fault Tolerance in Spark workloads.:* Our architecture offers fault tolerance for Spark workloads, since all intermediate data are stored remotely and, thus, each Spark workload turns into a serverless job where the execution of Spark Workers is ephemeral. We estimate that the approximate additional time that is required on a specific stage of a Spark job without Cherry is given by (1). In this respect, $T_{Spark}$ is the additional time needed, $p$ is the percentage of the completed tasks in the stage, $t$ is the total number of tasks of the current stage, $c$ is the completion time for a task, $e$ is the total number of executors in the Spark cluster and $l$ is the number of lost executors when a Spark Worker crashes. With Cherry, the additional re-computation overhead is constant, and equal to the completion time of the tasks that were running at the failure time, which is $c$ (i.e., $T_{Cherry} = c$).

$$T_{Spark} = \frac{ptl}{e(e-l)} * c \tag{1}$$

In order to showcase this feature we made the following experiment. We run our shuffle synthetic workload and created 20GB with 1000 mappers/reducers and a block size of 20KB. Furthermore, we killed a Spark Worker Pod when different percentages of the entire map stage have been completed, and compared the additional required re-computation time as well as the cloud resource utilization costs for the re-computation period for the tasks of which the shuffle data were lost in the vanilla Spark vs Cherry. Figure 6 illustrates the results of our experiment, where the solid line with triangles and the left y-axis depicts the extra time whereas the dashed line with X-spots and the right y-axis depicts the extra cost of native Spark compared to Cherry respectively.

Regarding execution time, Cherry's stateless architecture requires only the re-execution of the task that was running when the failure occurred, achieving a minimal constant overhead of around c=25 seconds in our experiment as baseline. On the contrary, with vanilla Spark, all the tasks that were computed by the killed executors by the time that the failure occurred have to be re-executed. For instance, in the case where a Spark Worker Pod failed at 80% of the map stage, Spark required 9X more time compared to using Cherry (rightmost triangle).

When it comes to re-computation costs, using the prices of [21], with 10 Spark Workers and 10 Cherry Pods running on an c1.medium EC2 instance each, we computed the total costs with native Spark vs Cherry as baseline, based on our previous time measurements. Similar to the time case, for instance, in the case where a Spark Worker Pod failed at 80% of the map stage, Spark was almost 5X more expensive compared to Cherry (rightmost X spot).

It is depicted that according to different values of the parameters on (1), there will be divergent additional overhead. For example, in one of our cases with $p$=40%, $t$=1000, $c$=25 seconds, $e$=10 and $l$=1, the additional time is $T_{Spark}$=111.1 seconds, which is 4.5X more compared to the baseline. Another theoretical example is as follows: for $p$=75%, $t$=4000, $c$=500 seconds, $e$=20 and $l$=2, with Cherry we will need only $T_{Cherry}$=500 seconds, while with (1) that results in $T_{Spark}$=8333.3 seconds, which is 16.6X more additional time. Therefore, Cherry's disaggregated architecture and serverless manner of execution of Spark workloads achieves great fault tolerance regarding Spark in real production environments with minimal additional overheads.

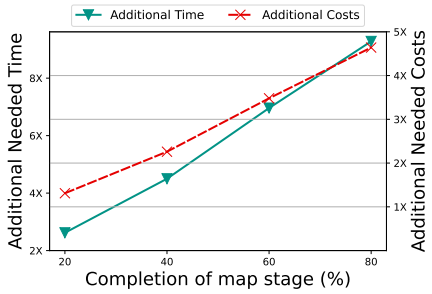


Fig. 6. Additional Vanilla Spark map stage completion time (left y-axis) and resource cost (right y-axis) vs Cherry for various % of map stage completions

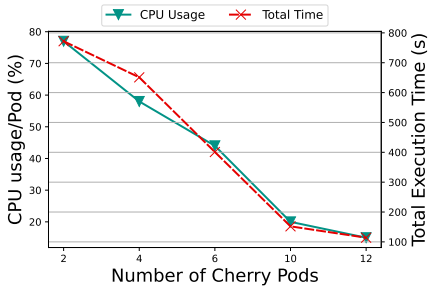


Fig. 7. CPU usage per Cherry Pod and total execution time while varying the number of them in the cluster.

*3) Scalability and Resource Efficiency:* Cherry's architecture enables its seamless scalability in a Spark cluster. Moreover, is has a low resource usage, both on CPU and memory end. To illustrate that, we run our synthetic workload with different number of Cherry shuffle services available within the cluster and 10 Spark Worker Pods, and monitored the CPU usage of each Cherry Pod, when the shuffle block fetch operation of the reduce stage takes place , as well as the total execution time of the workload. Figure 7 shows the experiment results. As the number of Cherry Pods is increased, the latter uses less amount of a CPU core and the total completion time is reduced drastically. This happens since the amount of requests from executors that have to be handled per Cherry in a certain time frame is decreased, and Cherry serves the shuffle blocks faster. Additionally, when the cluster has 10 Cherry shuffle services, each one uses about 20% of a CPU core. We, also, monitored the average CPU utilization of the Spark Workers with vanilla Spark and measured that they need about 35% of a CPU core. This is because the Worker's executor has to fetches its local shuffle data for its respective assigned tasks, as well as its ESS has to respond to requests and serve the shuffle data accordingly.

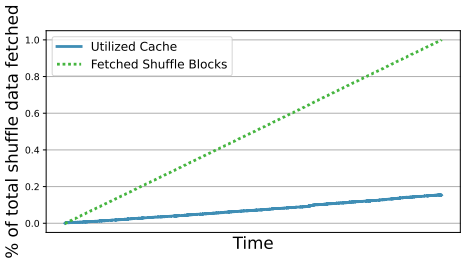


Fig. 8. Normalized Cherry Cache Usage compared to fetched shuffle blocks by executors through time.

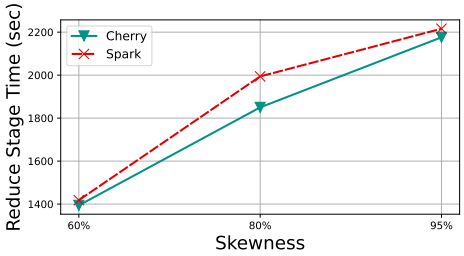


Fig. 9. Spark Performance with Cherry on skewed data vs Vanilla Spark.

As far as memory consumption is concerned, we measured how cache is utilized in all Cherry services for shuffling 50GB of data. Figure 8 shows how cache utilization changes through time, compared to the normalized shuffle blocks that are fetched by executors in the workload. Since Cherry caches and evicts immediately the blocks that are fetched from the Spark executors, we can see that the cache consumption increases slowly, compared to the intermediate data that are served, and ends up at about 18% of the shuffle data fetched. This slow increase is due to the blocks that are cached but are not fetched, since they are located locally in their respective executor. The percentage of these data depends on the number of Spark workers and Cherry services in a Spark cluster. Also, through our measurements, we found out that only 10% of the total shuffled data at most remains in the total cache of all Cherry Pods after a job completion. All of these shuffle blocks are evicted at the end of of the Spark job to retrieve the maximum available memory for upcoming workloads.

*4) Data Skews:* Data skews and task stragglers can severely impact the performance and completion time of jobs in large-scale analytics workloads. The process of dealing with these is of vital importance, and there is recent work that emphasize on that [22], [23]. Although this area is not our main focus in this work, we wanted to showcase how Cherry performs and copes with skewed data in a Spark job.

In order to examine this challenging aspect, we created a synthetic workload that requires 20GB of data shuffling with 10KB shuffle block size and allows the tuning of the percentage of the data skewness. Consequently, we executed it in on our aforementioned Spark cluster. Figure 9 shows our results. We can see that Cherry's distributed architecture and caching policy achieves relatively faster completion time of the reduce stage time. More specifically, Cherry has from 2% to 8% better performance, with the best score being on 80% skewed data. Thus, we believe that on large-scale workloads

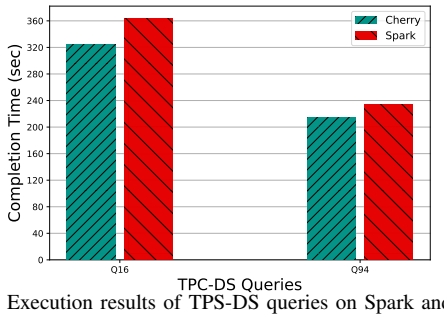


Fig. 10. Execution results of TPS-DS queries on Spark and Cherry.

Cherry's features can improve the data shuffle operation and reduce the job completion time, even when there are great data skews.

### C. Real Workload Evaluation

We further continue on evaluating Cherry on a realistic TPC-DS benchmark workload [24] against Spark with ESS. This is a decision support benchmark that models a general decision support system of a retail product supplier and includes a wide variety of SQL queries, such as ad hoc, reporting and iterative queries. For our evaluation, we use the same hardware resources and number of Pods in our cluster as used in our synthetic workloads. Additionally, we generated TPC-DS data with a scale factor of 100, which corresponds to all its respective tables adding up to a total input size of 100GB data, and stored them in HDFS.

Figure 10 illustrates the completion time of the total execution of the selected TPC-DS queries on Spark with ESS vs Spark with Cherry. Both executed queries, Query 16 and Query 94, are general report type queries that combine different tables and include shuffle-heavy operations. To ensure a realistic execution on a typical multi-node setup as described in [11], we selected a high number of partitions for the input data so as to assure that we will have small shuffle block average size, below some KBs, throughout the workload, since we can't explicitly determine the size of intermediate data in a complex job later in the DAG because it depends on the dataset size, data cardinality/selectivity, job transformations etc. For Q16, Cherry manages to reduce its completion time from 364 seconds to 324 seconds, which is 11% completion time reduction. Moreover, for Q94, Spark with ESS requires 8.5% more time to complete than Spark with Cherry.

## VI. Related Work

**Shuffle Operation Optimization:** Data analytics systems such as MapReduce and Spark have been studied both by industry and academia. *Magnet* [11] and *Riffle* [12] modified the Spark ESS and tried to solve the efficiency issues created by disk I/O bottlenecks that occur when fetching of small sizes of shuffle data is required by achieving sequential disk reads of bigger sized shuffle chunks. *Magnet* suggested pushing shuffle files from Spark executors to the shuffle services and merging them per shuffle partition. With *Riffle*, there is a shuffle service running on each physical node and tries efficient block merging by pulling the intermediate data files from map tasks and executing their merging. However, there

is no fault tolerance provided in case of node crashes since merged blocks are located on compute nodes.

Apache *Crail* [13] achieves high performance by using specific storage resources and networking and can be used to execute data analytics workloads in a disaggregated architecture.Nevertheless, their hardware is specialized and there is no addressing in the challenges of small intermediate blocks in shuffle operations. Another research work [25] focuses on allowing compute and storage engine disaggregation as well as on addressing the existing Spark issue of fault tolerance by creating a remote manager that keeps all intermediate data from workloads to its file system. However, no optimization on the operation of reducers reading shuffle data is mentioned.

*Cosco* [14] focuses on aggregating shuffle intermediate blocks with buffers and uses an existing remote file system for storing these files. *Sailfish* [15] and *iShuffle* [16] aim at optimizing the shuffle phase operation on Hadoop MapReduce workloads. *Sailfish* extends an external filesystem and aggregates intermediate shuffle blocks of jobs and, thus, decreases the number of block fetches by reducers. However, it compromises fault tolerance due to chance of corrupted aggregation files. Both *Cosco* and *Sailfish* works rely on external storage systems to operate as shuffle services. *iShuffle* pushes mapper shuffle data to the reducers but does not enhance shuffle performance. Another relative work presents *Flint* [17], which is a rewrite of the execution engine of Spark and exploits AWS Lambda [26] but is focused on only the pySpark interface.

**Serverless Architecture:** When it comes to research and literature, there is plenty of public work relative to serverless platforms and architecture. *Pocket* [27] is an elastic data storage that can maintain ephemeral data. It is built on top of Apache Crail [13] and AWS Lambda is used as a serverless compute engine. However, they do not address the challenge of storage node failures and argue that fault tolerance is not mandatory. Another recent work is *Shredder* [28], which examines multi-tenant isolation on serverless environments and pushes functions into storage. However, it is executed in a single node and does not address any fault-tolerance feature.

*Cloudburst* [29] is a stateful FaaS platform that is built on top of *Anna* Key-Value Storage [30], [31] and can efficiently leverage its integrated caching capabilities in order to process key-value storage objects with minimum latency. *Cloudburst* runtime can also autoscale independently from Anna and achieves disaggregation. Nevertheless, there are no guarantees relative to failures on the compute tier, and the whole DAG of a job will be re-executed. *FaaSFS* [32] is a file system for stateless cloud functions that utilizes a familiar POSIX API, reaching performance close to what a local file system achieves, but is not benchmarked on data analytics workloads.

## VII. Discussion

The original MapReduce framework and all of its descendants employ an intermediate shuffling mechanism that, at its simplest form, utilizes the local filesystem of the participating executors. All shuffling approaches, either local or remote, require a persistent storage and an addressing mechanism to store and retrieve intermediate data: storing is done by utilizing POSIX read/write system calls in the local case or RPC Put/Get calls in the remote case and addressing is done through a combination of executor IP address and filename (reduce bucket), in order to uniquely identify the specific intermediate data that needs to be fetched.

These characteristics (i.e., shuffle storage and addressing) are being taken into consideration by Cherry throughout its design, and a lean approach with well-defined external system interactions is followed, to facilitate its generic applicability: In Figure 3 we notice that Cherry interacts with the external big data processing system only in its Put/Get and task info creation steps, thus requiring system specific code to be implemented only there. In fact, code changes to the big data processing system are required only in the respective steps: Cherry requires Spark to adapt only its Put (step 2) call, and task info call (step 4), whereas the adaptation is minimal with around of 500 lines for the configuration of API messages between Cherry and Spark (i.e., Cherry code inside Spark). On the other hand, Cherry required around of 500 lines of Spark-specific code in its implementation for processing the aforementioned messages, as well as caching the required blocks (step 5) and responding to Get requests of Spark executors (step 7).

Generic caching policies (for instance, LRU, NFU, etc.) are employed when the data access pattern is not known beforehand, something that we overcome in our case: by exploiting the scheduler task info we know both the exact blocks that are going to be requested and the time that this is going to happen. Therefore, the proposed look-ahead block-based caching policy comes as a natural optimization that Cherry can achieve by utilizing its two distinguishing characteristics, namely the external shuffle storage service (through disaggregation) and the exact data access pattern knowledge. Regarding the caching policy generic applicability, since all those frameworks employ a task scheduler that assigns data to executors, we can easily extend Cherry by consuming similar system-specific scheduling info.

## VIII. Conclusion

We present Cherry, a distributed shuffle service for large-scale analytics workloads that leverages a look-ahead caching policy on task-level that efficiently improves the I/O bottlenecks on the shuffle operation of small shuffle block sizes. Cherry is disaggregated from the Spark components and can store all intermediate data from workloads seamlessly, transforming Spark into a pure serverless analytics engine. It also achieves fault tolerance in case of node failures and has low resource footprint. Through our experiments, we show that Cherry overcomes the challenges of shuffling in Spark and improves the required time of shuffle fetching in analytics jobs.

## IX. Acknowledgement

This paper is supported by EU's Horizon 2020 Framework Programme - GA Number: 861377, project IW-Net.

## REFERENCES


[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte *et al.*, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1802–1813.

[5] S. Krishnan and J. L. U. Gonzalez, "Google cloud dataflow," in *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 255–275.

[6] Apache spark, retrieved sep. 1, 2021. [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html#configuration-and-setup

[7] Apache software foundation, retrieved sep. 1, 2021. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html

[8] Z. Wang. retrieved sep. 1, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-31%3A+Pluggable+Shuffle+Service

[9] T. P. Foundation. retrieved sep. 1, 2021. [Online]. Available: https://prestodb.io/docs/current/admin/exchange-materialization.html

[10] Google. retrieved sep. 1, 2021. [Online]. Available: https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline/#dataflow-shuffle

[11] M. Shen, Y. Zhou, and C. Singh, "Magnet: push-based shuffle service for large-scale data processing," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3382–3395, 2020.

[12] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[13] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance i/o architecture for distributed data processing." *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, 2017.

[14] D. borovsky. [Online]. Available: https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service

[15] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012, pp. 1–14.

[16] Y. Guo, J. Rao, D. Cheng, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *IEEE transactions on parallel and distributed systems*, vol. 28, no. 6, pp. 1649–1662, 2016.

[17] Y. Kim and J. Lin, "Serverless data analytics with flint," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 451–455.

[18] Google. retrieved sep. 1, 2021. [Online]. Available: https://kubernetes.io/docs/home/

[19] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[20] Amazon web services, retrieved sep. 1, 2021. [Online]. Available: https://calculator.aws/#/createCalculator/EBS

[21] Amazon web services, retrieved sep. 1, 2021. [Online]. Available: https://instances.vantage.sh/?selected=c1.medium

[22] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 185–198.

[23] C. Wang. retrieved sep. 1, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/spark-sql-adaptive-execution-at-100-tb.html

[24] R. O. Nambiar and M. Poess, "The making of tpc-ds." in *VLDB*, vol. 6, 2006, pp. 1049–1058.

[25] C. Guo. retrieved sep. 1, 2021. [Online]. Available: https://databricks.com/session_eu19/improving-apache-spark-by-taking-advantage-of-disaggregated-architecture

[26] Amazon web services, retrieved sep. 1, 2021. [Online]. Available: https://aws.amazon.com/lambda/

[27] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 427–444.

[28] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 1–12.

[29] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *arXiv preprint arXiv:2001.04592*, 2020.

[30] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[31] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," *The VLDB Journal*, vol. 30, no. 1, pp. 25–43, 2021.

[32] J. Schleier-Smith, L. Holz, N. Pemberton, and J. M. Hellerstein, "A faas file system for serverless computing," *arXiv preprint arXiv:2009.09845*, 2020.