Journal of Grid Computing (2021) 19: 39
https://doi.org/10.1007/s10723-021-09580-0

A Reliable Large Distributed Object Store Based Platform for Collecting Event Metadata

Álvaro Fernández Casaní · Juan M. Orduña · Javier Sánchez · Santiago González de la Hoz

Received: 23 December 2019 / Accepted: 20 July 2021 / Published online: 30 August 2021

Abstract The Large Hadron Collider (LHC) is about to enter its third run at unprecedented energies. The experiments at the LHC face computational challenges with enormous data volumes that need to be analysed by thousands of physics users. The ATLAS EventIndex project, currently running in production, builds a complete catalogue of particle collisions, or events, for the ATLAS experiment at the LHC. The distributed nature of the experiment data model is exploited by running jobs at over one hundred Grid data centers worldwide. Millions of files with petabytes of data are indexed, extracting a small quantity of metadata per event, that is conveyed with a data collection system in real time to a central Hadoop instance at CERN. After a successful first implementation based on a messaging system, some issues suggested performance bottlenecks for the challenging higher rates in next runs of the experiment. In this work we characterize the weaknesses of the previous messaging system, regarding complexity, scalability, performance and resource consumption. A new approach based on an object-based storage method was designed and implemented, taking into account the lessons learned and leveraging the ATLAS experience with this kind of systems. We present the experiment that we run during three months in the real production scenario worldwide, in order to evaluate the messaging and object store approaches. The results of the experiment show that the new object-based storage method can efficiently support large-scale data collection for big data environments like the next runs of the ATLAS experiment at the LHC.

Keywords Grid computing · Hadoop file system · Object-Based storage

Á. F. Casaní (✉) · J. Sánchez · S. G. de la Hoz
Instituto de Física Corpuscular (IFIC), Universidad de Valencia and CSIC, Valencia, Burjassot, Spain
e-mail: alvaro.fernandez@ific.uv.es

J. Sánchez
e-mail: javier.sanchez@ific.uv.es

S. G. de la Hoz
e-mail: santiago.gonzalez@ific.uv.es

J. M. Orduña
Departamento de Informática, Universidad de Valencia, Valencia, Spain
e-mail: Juan.Orduna@uv.es

1 Introduction

The Large Hadron Collider (LHC) is a particle accelerator located at CERN near Geneva, at the border of Switzerland and France, with a circumference of 27 km and placed in a tunnel 175 meters below ground. Modern High-Energy Physics experiments at CERN, and in particular the ATLAS experiment [1], produce petabytes of data per year which have to be catalogued in order to be available in a proper way to users and applications. ATLAS finished its Run-2 phase (2015-2018) detecting particle collisions, or events, at

rates as high as 40 MHz. A multi-level trigger system selects events of interest for offline analysis at rates up to 1 kHz. These events are recorded in files and they are processed later in different formats usually multiple times. This reprocessing is carried out by the collaboration of 3000 scientists from about 180 institutions all around the world. For example during year 2017, 10 billion real events were recorded in 7 million files, and then reprocessed to reach 270 billion event record entries in 38 million files distributed in the Grid. Additionally every year billions of events are simulated to perform comparisons between data and theoretical physics models, as well as for software and data validation studies. The EventIndex [2, 3] is a metadata catalogue at the event level which tries to exploit technologies such as Hadoop [4] as a backend storage. After a prototype stage, it has been running in production since the start of LHC Run 2 in 2015, indexing all produced data which is though to be of interest for physics analysis. The distributed data collection task of the EventIndex project [5] is responsible for retrieving the relevant information of all the events from all the permanent dataset files, and transferring it to the Hadoop cluster at CERN, where it can be structured and aggregated by the Hadoop core task. Thus, the EventIndex project can be considered as a representative example of real grid systems in large distributed infrastructures generating big data [6–8]. The original EventIndex data collection design follows a producer/consumer architecture as depicted in Fig. 1, where the producers are processes running at distributed centers, and the consumers are processes running at CERN central facilities, communicating with an ActiveMQ [9] based messaging infrastructure.

The original data collection system was designed with scalability in mind, but the current nature of the segmented payloads caused the messaging brokers to become a system bottleneck in some situations [10]. Head-of-line (HOL) blockings [11] arise when large backlogs of messages destined for one group, block the processing of messages corresponding to other groups. As a result, messages accumulate in the broker, which does not correctly distribute the workload to consumers. In these cases, the addition of more consumers for a particular broker does not improve the throughput. Adding more brokers alleviates the problem, but as the brokers are a more expensive resource

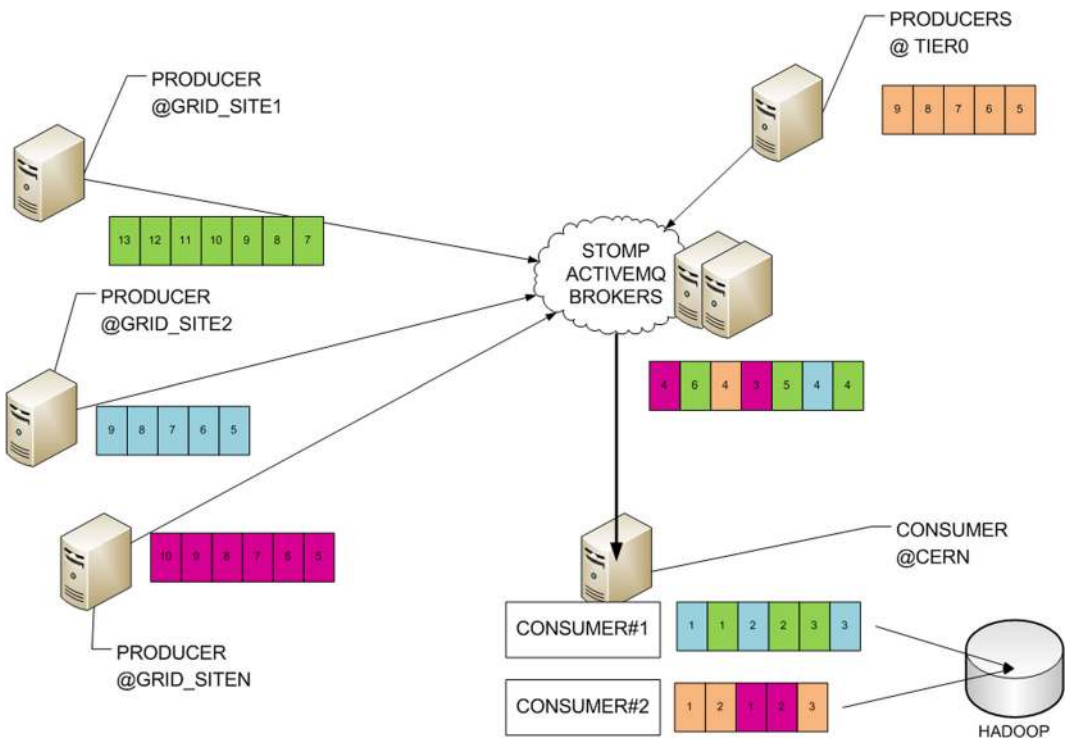

Fig. 1 Original EventIndex messaging data collection architecture

in terms of hardware and memory, this is a limiting factor.

This situation did not become a problem yet, since the current steady rates are still low enough to allow the backlog produced by the peaks to be absorbed at slower rates. However the amount of data produced by ATLAS is expected to increase in the future with a prediction of 35 billion new real events per year in Run 3 (2021-2023), and 100 billion new real events per year in Run 4 (2026-2029). For simulated events the numbers are even higher, with 100 billion events/year in Run 3, and 300 billion events/year in Run 4. Additionally all this data will be reprocessed, increasing by several orders of magnitude the number of event entries and the rates of messages to be handled to unmanageable levels. For this reason, several aspects from the distributed data collection system of the EventIndex project should be improved, in order to guarantee an efficient handling of all the indexed data in the next ATLAS run.

The main contribution of this paper is the redesign of the data collection system of the ATLAS EventIndex project, addressing the scalability issues and reducing the overall system complexity. In order to solve these weaknesses, we proposed and implemented a different approach, using an object storage system (OBS) [12] for storing temporarily the payload, and sending only a reference to be later used for consumption. With this approach there is no need for payload segmentation, as the OBS allows storing all the payload from a producer in a single object. Bigger payloads and binary encoding adds the possibility of applying different schemas and achieving better data compression ratios. We therefore avoid large segmented payloads at the messaging broker queues to be sequentially consumed. OBS do not block the consumption by different consumers, so we can improve workload distribution and eventually the scalability of the system. The usage of a temporary storage like an OBS also allows the dynamic selection of data, eliminating the need to consume duplicated produced data. OBS systems are a popular technology these days, but its usage for pull-model scenarios with large payloads, and dynamic data selection like our application has not been widely tested in large scale production grids, to the best of our knowledge.

The new approach results in performance improvements, resource consumption reduction, and better final user experience, and might be useful for other large distributed grid systems with similar payloads and characteristics.

We make a comparative evaluation of both messaging and object store approaches in our real production environment. The EventIndex production was run during three months worldwide evaluating both methods in parallel, indexing 60 billion events stored in more than 10 million files, summing up 17 petabytes of input data. The results show that the object-based storage (OBS) method seems an appropriate option for large grid systems generating big data like the one required for the next run of the ATLAS experiment at CERN.

The rest of this paper is organized as follows: Section 2 relates the motivation and the design of a new object-store based data collection system. Next, Section 3 shows the comparative performance evaluation of both transport methods on a real production scenario. Finally, Section 4 provides some concluding remarks.

2 A new design for distributed data collection

The congestion problems and weaknesses mentioned in the previous section with the messaging approach [10] limit in practice the performance and scalability of the data collection system required for the next runs of the ATLAS experiment.

Based on the experience acquired by running the ATLAS EventIndex data collection for the last years, the following weaknesses that should be addressed were identified:

- **Complexity** Messaging systems are designed to handle a large number of small messages, but our typical payload consists on large data files that have to be divided into smaller messages. This segmentation and re-assembly procedure is complex, and it has an effect on the brokers and consumers performance.
- **Scalability** The current messaging data collection system is not able to scale up to the event processing rates that will come in the following years. We need a system that does not degrade with the occurrence of slow consumers, and that does not produce backlogs.
- **Performance and resource consumption** Although the general performance of producer

and consumers is good, the complexity of payload segmentation and reassembly requires high rates of CPU and memory consumption. Also, the usage of a text protocol limits the payload encoding and compression factors. Additionally the HDFS backend where the data is written by consumers is not used in the best possible way, as the number of written files is very high and their size small, compared with the ideal bigger files for Hadoop. There are compaction and grouping procedures in place, and extra steps needed to validate and remove the invalid data, because some jobs can be run twice (producing duplicates) due to the nature of the grid.

– **User experience** The traversal time since data starts to be indexed until it is finally available for users can be improved. The issues explained above impose limiting factors and extra steps that could be avoided, making final validated data available in a faster way.

In order to tackle these challenges, we first explored other alternatives to convey the EventIndex data with different transport methods for our distributed data collection task. The search started with other messaging systems like RabbitMQ [13]. Also other big data streaming systems like Flume [14] and Logstash [15] were considered, but these systems are log-oriented, which is not suitable for our application. We also reviewed Kafka [16] as a promising alternative, as it is widely used in the context of transporting and loading data into Hadoop. Kafka is shown as a more performant option with higher throughput than ActiveMQ and RabbitMQ [16] when assuming at-most-once delivery semantics, and implements interesting features for our use case like longer term message store or message replay capabilities. On the other hand requiring more strict semantics like in-order delivery of messages imposes a single partition, and at-least-once delivery semantics reduces performance up to a 75% [17]. Yet there are other problems that make Kafka not suitable for our use case. It does not currently support transactions as required to assure atomic collection of related data, and in case of problems, messages might be replayed and a deduplication mechanism is needed.

The ATLAS EventIndex project built the data collection system based on previous experiences with these messaging systems in large distributed infrastructures. However, the future EventIndex project imposes tougher reliability requirements, pseudo-realtime restrictions in delivery, and a higher amount of data per producer, which builds a different scenario. We need a system that allows long-running producer tasks to be decoupled from consumers, and that handles reliably bigger payloads per producer (up to the order of hundred of megabytes). This payload has to be consumed in an atomic manner, and be temporarily available until is consolidated in the final data backend. And of course we need a good control over scaling and performance, that has reached its limits with the current messaging system.

In general messaging systems are designed to handle a large quantity of small messages, and although Kafka and recent versions of ActiveMQ (Artemis release), can be configured to handle some large messages in case of necessity, its performance degrades over time, especially with the advent of slow consumers. When we also impose the previous restrictions we realize we need another approach.

There are also other methods to convey big data in large distributed systems. For example, the storage at large production grids like the WLCG [18] is based on tape and disk resources in each distributed grid site, and a global data management system [19].

These restrictions led us to consider a method based on a completely different approach. Instead of chunking and submitting the payload with a messaging system, we considered the use of an object-based storage (OBS) to temporarily store the payload, and submit a small message with a reference to be used by the consumers to retrieve the data. With this approach we can avoid the need for payload segmentation and the partitions (MessageGroups) that cause the bottlenecks.

One of the reasons for considering an OBS system among other options is the satisfactory previous experience using this kind of system. The ATLAS team had some experience using OBS systems for storing some events that are produced on non-grid resources like HPCs ('hole filling'), commercial clouds (dynamic market based 'spot' pricing) and volunteer computing (no resource availability guarantees) [20].

There are some features in OBS systems which make them an interesting option. Unlike files in filesystems, OBS systems store data in a flat structure [12], so there are no folders nor hierarchy among them, only a pool of objects represented by their object

identifiers. So in terms of scalability it is a good candidate for storing a high quantity of objects like in our use case. Also, arbitrary metadata can be assigned to any object, providing more flexibility than classical file systems. Additionally, the metadata directly live in the object, rather than for example in a separate i-node. There are examples of metadata for supporting data management object movement from one storage site to another, or associating tags relative to the specific object content [21]. In the EventIndex case, we can store specific EventIndex metadata that is useful to identify the event information that the object includes, like the number of events it contains, the size, the producer process identifier and the grid job that executed it, as well as some other data. The third key point of OBS systems is the existence of a globally unique identifier of the object in order to be found in a distributed system. In this way, it is possible to find the data without having to know the physical location of these data, even if the storage is spread among different parts of the world. Object-based storage generally differs from block storage, as it generally does not provide the capability of incrementally modifying one part of the object. Objects have to be manipulated as a whole unit, requiring the entire object to be accessed, updated, and/or rewritten entirely, thus avoiding the complexity added by the message segmentation in the previous approach. The main benefit is scalability, solving problems of data growth with the easy management of adding additional nodes. The flat namespace organization of data and metadata colocation within the object is by design appropriate for this capability. Since data are agnostic to location, OBS also provides greater flexibility. Moreover, it also provides increased levels of resiliency, as multiple copies of data exist over a distributed system. In the case of a node failure the data can still be made available, in most cases without the application or the end user ever being aware of the failure. All these reasons led us to the selection of the OBS system.

For the EventIndex this approach means a change in the data collection model, from a push model where all the data flows through the producer-broker-consumer chain to the Hadoop backend, to a pull model where the consumers are signalled what data to retrieve from the OBS. This pull model also allows the dynamic selection of which data are finally stored at the Hadoop servers. This is specially important in our case, because there can be duplicated produced data, due to grid job failures or restarts in our extremely distributed infrastructure. With the messaging push model, all data, including duplicates, are stored in Hadoop, and extra cleaning-up procedures are needed to remove and validate these data. With the pull model all data, including duplicates, are temporarily stored at the OBS, but only valid data are consumed and stored in Hadoop, sending less data than the messaging approach and avoiding the need of extra, expensive cleaning steps. There are other advantages with the pull model, like the possibility of consuming the same produced data several times. This issue is useful to overcome spurious consumption problems for example due to a consumer machine crash. This option was not possible with the messaging push model, where the data could be consumed only once.

Many changes are needed in order to adapt a new transport protocol based on OBS to the EventIndex system. The basic producer/consumer architecture with the same elements is still valid in this approach, due to the distributed nature of the EventIndex production. However, the payload is not conveyed using a messaging system with brokers, but temporarily written by the producers into an OBS system.

Figure 2 represents the new producer/consumer based architecture that we need due to the distributed nature of the EventIndex production. Compared with the original messaging architecture depicted in Fig. 1, now the payload is not conveyed using a messaging system with brokers, but temporarily written by the producers into an OBS system. The supervisor entity substitutes the validator, enriched with more intelligence needed to orchestrate the procedure. This is needed because we change the data consumption to a pull model where the consumers are informed by the supervisor about the data they have to retrieve from the OBS. The supervisor is in charge of selecting the valid produced information and signaling consumers to retrieve the appropriate data from the OBS system. The communication for these entities is done with control and statistic messages similar to the ones from the messaging scenario, so we still use queues from the brokers to distribute the processing messages among different consumers.

Figure 3 shows the resulting sequence diagram. The producer now puts all the event index data in a OBS, without dividing the payload in various messages, and when it is done it sends a control and statistic messages to the statistics queue of the broker.

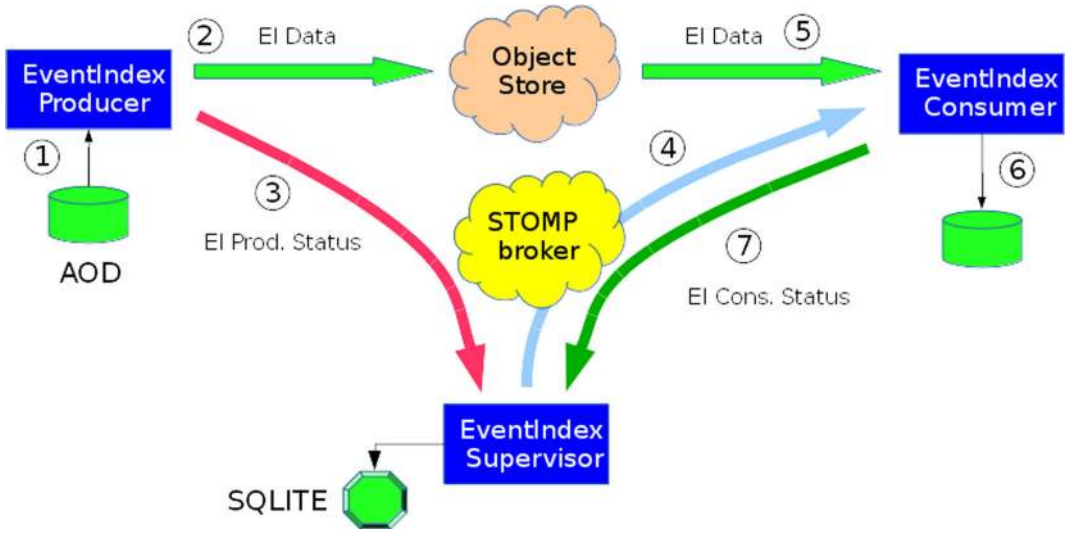

Fig. 2 New EventIndex object store grid data collection architecture

The previous phase is repeated continuously by every producer running on the grid or at CERN, with the supervisor tracking all the data produced. Some datasets contain hundreds of files with millions of events, so multiple producers might run in parallel until all the related data is indexed. The supervisor entity also takes into account the possibility that some fraction of the event processing does not reach its final state, as it was done with the validator in the messaging scenario. Now the difference is that this partial data is not being continuously pushed to the consumers.

When reaching a desired processing granularity (for example indexing all the data from a dataset), the supervisor signals the consumer with a control message as can be seen in Fig. 3, which contains all the information needed to retrieve the data by the consumer. This allows a consumer to consolidate information in a single step, writing a unique file in the HDFS filesystem, instead of having multiple files written by several consumers like in the messaging scenario. Table 1 shows a summary on the differences in the new object store pull-based model compared with the previous messaging push-model.

Two key differences should be noted in the new approach. First, the entire payload from a given producer can be potentially stored within a single object regardless of its size. In this manner we can avoid payload segmentation and the need for partitions and transactions, so we reduce overall system complexity and the burden on the messaging brokers. With coarser

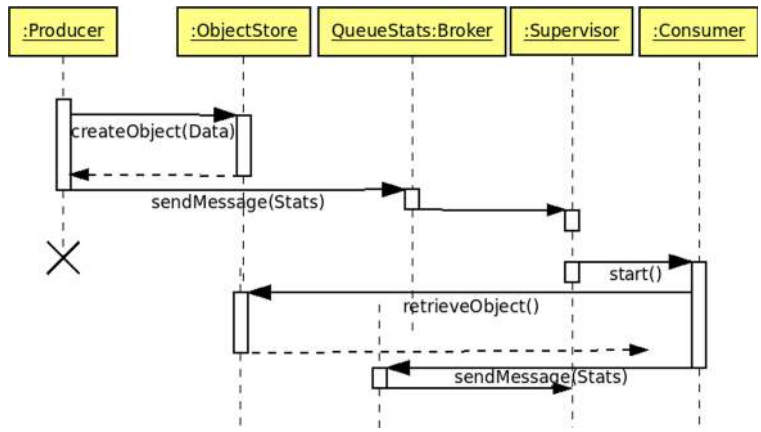

Fig. 3 Object store scenario sequence diagram

Table 1 Summary of concepts of the messaging push model versus object store pull model

Concept		*Messaging push-model*	*Object Store pull-model*
Data			
	Payload	many messages per input file	single object per input file
	Segmentation	transactions group messages	not needed
	Reconstruction	complex	not needed
	Encoding	textual	binary
	Compression	low	high
Workload distribution			
	Producer-Consumer	1-to-1	1-to-many
	Production	a transaction is reconstructed by a single consumer	multiple produced objects can be retrieved by different consumers.
	Consumption	FIFO (message queues abstraction)	arbitrary
		assure all data is consumed complete and in order	consumption out of order possible
	Blockings	slow consumers can block others.	no blockings, more scalable
	Lifetime	short (messages removed when consumed)	definable (object can be retrieved multiple times in case of failure)

data granularity we can also apply different encodings and achieve better compression ratios.

Second, the behavioural change from a push model to a pull model. This model allows to use the OBS as temporary storage from which only valid data is retrieved and consolidated into bigger, more suitable files in the HDFS filesystem. It also reduces the amount of data that is consumed, and thereby the network usage from the OBS to the final HDFS backend. We can also avoid extra and expensive cleaning tasks on the Hadoop cluster.

With the previous messaging approach, all the data should be consumed in FIFO order due to the queue abstraction on the brokers. In addition, data from a particular producer (multiple messages grouped by transactions) should be consumed by a single consumer, to assure data was completely retrieved in order. With the object store approach, object consumption does not need to be in order, and can be done by different consumers. Objects can be accessed multiple times if needed, as they have longer lifetimes than messages that disappear when retrieved from the broker.

3 Evaluation

To test the validity of the new approach and whether the new object store method can substitute the previous messaging-only approach, a scenario in real production was set up which collected data and statistics for three months.

Figure 4 illustrates the evaluation setup. In this scenario, both collection methods are run in parallel, but only replicating the essential parts that are needed. So for example there is a set of producers running worldwide, that index data only once per input file, but they send the same index data to both messaging and object store platforms. Then we have two sets of consumers, one for messaging and the other for the object store, each contacting its source of data and storing it in different areas of the backend HDFS filesystem. And we have two different supervisors to control both data collection paths, and that the overall procedure is correct.

The object store system (OBS) is implemented with Ceph [22, 23], installed at CERN and supporting the S3 interface [24] to store up to 2 PB. The messaging system uses six ActiveMQ brokers also at CERN

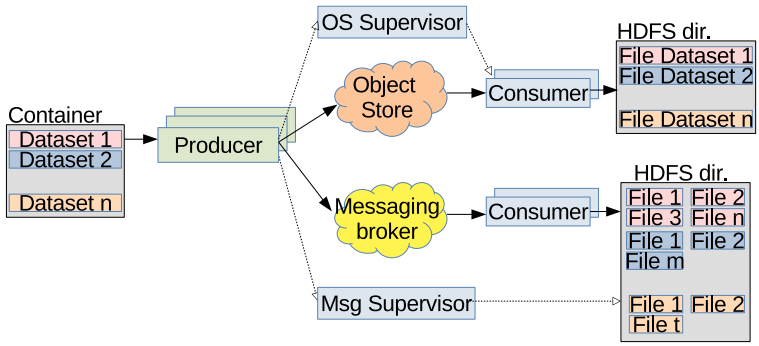

Fig. 4 Evaluation setup with messaging and object store in parallel

under a DNS alias to achieve high availability. When analyzing the input data it is worth noting how data in ATLAS are stored and logically grouped, to see how data are distributed to producers to be indexed, and how data are finally validated and made available to users. Basically events are stored in files, identified by a unique GUID (a global grid unique identifier). Datasets group several files that have related data, and are named following ATLAS conventions. For grouping datasets we use dataset containers, which are a generalization of the dataset concept. The overall picture is the following:

$$Container \supseteq Dataset \supseteq File \supseteq Event$$

The minimum unit for indexing data by the producers is the File, but usually the information in a single file does not contain all the required information for users, who are more interested in complete Datasets or more often Containers. So the minimum unit of information that is validated and made available for final users is a Dataset. Once validated and available in Hadoop, all the files from a Dataset are searchable by the users. Figure 4 illustrates how a Container with several datasets is indexed. Short lived producers are spawned with grid jobs, at the sites where the files reside. Files can be replicated at several grid storage elements, to achieve redundancy and high availability for some important or hot data, so the distribution of grid jobs with producers is rather dynamic. One producer may index one or more files from a dataset, so for big datasets there will be a high number of relatively short-lived producers. During this experiment scenario, a producer indexes input data only once, but the same index information is sent along 2 different paths, with different data encodings and different methods controlled with their respective supervisors. The path of the payload (the index data) is represented in the figure with solid black arrows, and the control messages among the different entities are represented with dashed arrows.

The messaging path can be seen in the bottom part of the figure: the producer sends the index data encoded with textual JSON representation, divided into smaller 10kB messages, to the messaging brokers using the STOMP protocol. For every original input file indexed, the producer sends its payload with a unique transaction, grouping several messages. A producer indexing several input files will establish several transactions with the broker, each containing probably hundreds of messages. After completing the last transaction, it sends a final control message to the messaging supervisor entity, with identifiers information including the files processes and the dataset they correspond to, along with some statistical information including the number of events processed,and messages and bytes sent.

The messaging consumers are constantly receiving messages from the brokers. Each original input file is assured by the broker to be assembled by a unique consumer, which will store that data in a single HDFS file. We can see in the figure that the n color-coded original files of a dataset, will be stored in n different files in HDFS. Then we can have other m files from a second dataset, and t files from another one, all residing in the HDFS directory named after the Container name. The last step is done by the messaging supervisor, when it has received all the control messages from producers and consumers to know that the data resides in HDFS files. Remember that data can contain duplicates, so this step performs asynchronous validation and cleaning of duplicated data, before making finally available to users.

The upper path of the figure represents the object store path, where a producer stores its index data in a single object in the object store. The approach is

different from messaging because we don't need to divide the payload into smaller chunks, and we can also use a binary compressed format based on Protocol Buffers [25], so we can produce more compact information stored as a single object. This means less information sent and easier management compared with the messaging approach. Likewise, when the producer ends storing its index data at the object store, it sends a control message with similar statistics to the object store based supervisor, including the object id where to find this stored index information.

The approach with the object store consumers is also different, since they are not constantly consuming data, but wait for a signal from the object store supervisor, pictured with a dashed line in the figure. This signal is sent from the supervisor when a Dataset is completely indexed and stored in one or more objects in the object store. Upon receiving the signal, the consumer retrieves all the needed objects from the OBS, and stores the index data in a single HDFS file. This is another difference in the philosophy of the approach, since we can see in the picture that the n input files from a Dataset are stored in a single HDFS file. If there are more Datasets belonging to a particular Container, they will be retrieved when indexed in the same manner, and stored in files in the HDFS directory named after the Container identification. This second approach is simpler because it uses fewer, bigger files in HDFS, and it does not have to run a final cleaning and validation step, since all the data read from the OBS and written in HDFS by the consumers are already validated online by the supervisor.

3.1 Results indexing a single dataset

As a first step to analyze the differences with both approaches, we study a concrete example of indexing a complete dataset within the previous scenario setup. We will check the information that we obtain from both paths, messaging and object store. Later on we will review global results for all the indexed datasets processed during the 3 months that experiment lasted.

Table 2 shows a summary of the input data and the results obtained taking into account the producers, consumers and the overall process.

Input Data This test processed a single dataset of real ATLAS experimental data taken in June 2017. This dataset contains just over 21 million events, divided into 1160 files summing up 6.229 TB of data, which is the input data to be indexed by the EventIndex producers.

Producers With this kind of dataset every producer indexes one input file, but we can see that eventually more producers were spawned. This means that 287 jobs failed and had to be restarted at some point, and some of them probably produced some duplicated indexed data. Our system is able to cope with the nature of the grid and takes into account duplicated data from restarted jobs, to achieve exactly-once processing semantics. In this scenario the producers send the indexed data to both messaging and object store systems. We can see that the data sent through messaging needed more than 1.5 million messages, and the same data was stored within 1447 objects in the object store (corresponding to one object per producer spawned). Since the data encoding is also different, we see that data size sent through messaging (15 GB) is higher than through object store (3.3 GB). While the STOMP messaging imposes a text format, we can use binary compressed format with the data sent to the object store, achieving better data size ratios. The indexing time lasted about 55.5 hours from the start time of the first job until the last job finished.

Consumers. Messaging consumers receive in 'real time' all produced index data, but object store consumers are signalled to receive only correct validated data. This means some extra objects are not accessed, reducing the amount of received data a 25% in this example (2.5 GB received compared with the 3.3 GB produced and stored in objects). Object store consumers spend less time accessing and consolidating data in the final HDFS backend, than the messaging ones. The bandwidth reading parameter represents the amount of data received, divided by the time required to perform this task. The bandwidth writing parameter represents the amount of data written in HDFS, divided by the time required to perform this task. Each set of consumers writes in its own HDFS area, but the object store consumers write a unique consolidated file compared with the thousands written by messaging consumers. The final metric to characterize both methods is the throughput, and in this example the object store consumers reach roughly 6 times more consolidated events per second.

Table 2 Results of indexing a single dataset comparing both messaging and object store approaches

Concept	*Messaging*	*Object Store*
Input Data		
Input Dataset	data17_13TeV.00327636.physics_Main.merge.AOD.f838_m1824	
Total Files	1160	
Total Events	21103653	
Total Size	6.229 TB	
Producers		
Instances (short-lived)	1447	
Index results	1447 (287 duplicated)	
Events indexed(duplicated)	26287826 (5184173)	
Time spent	55.5 hours	
Transport method	1515131 messages (text)	1447 objects (binary)
Index size	15 GB	3.3 GB (Compressed)
Consumers		
Instances (long-lived)	6	6
Files received (Duplicated)	1447(287)	1160(0)
Events received (Duplicated)	26287826 (5184173)	21103653(0)
Receiving method	1515131 messages	1160 objects (287 not accessed)
Data received	15 GB	2.5 GB (Compressed)
Time spent	244 min	34 min
Reading rate (kB/s)	1098 kB/s	1302 kB/s
Throughput (events/s)	1793 events/s	10344 events/s
Output data (HDFS)	1447 HDFS files / 11.2 GB	1 HDFS file / 9 GB
Writing rate (kB/s)	627 kB/s	4464 kB/s
Output Index Data		
Validation Method	check and remove duplicates	not needed
Traversal Time	69 hours	56.1 hours
Consolidated Data (HDFS)	1160 Files / 9 GB / 21103653 events	1 File / 9 GB / 21103653 events

Validation and traversal time The total time for validating data until it is available (traversal time) is shorter with the object store approach. The validation is done differently depending on the approach taken. With messaging, all data is received constantly by the consumers and written in HDFS, so it requires an asynchronous final step done by the supervisor, when all data has been received, to check if all valid data is available, and to clean duplicates. On the other hand, with the usage of an intermediate storage like the object store, the supervisor can make the validation online and dynamically select and signal the consumers to read only valid data from the object store in a single step, writing a unique HDFS valid file. This also avoids the need to read again the HDFS files and delete the unnecessary ones. The producer times are the same for both paths (in this example, around 48 hours until the last data is indexed), so this part of the process is common. Messaging consumers write 1447 HDFS files as we have seen in the previous paragraph, and in this validation step done by the supervisor, it will check the files and remove the 287 extra HDFS files that contain duplicate data. This final validation step ends with 1160 HDFS files summing up 9 GB of index data, containing the 21103653 events. On the other hand the object store consumers are signalled to read only the 1160 valid objects in object store, and write a unique HDFS file with the 21103653 events. As the file format in HDFS is the same, it also accounts for 9GB of index data. The traversal time,

from the start of indexing time until it is available for users, is 69 hours with the messaging approach, and 56.1 hours with the object store approach. So we can see that with the new object store approach, consumers spend less total processing time, and the validation procedure is improved compared with the previous one, using less resources and making data available to final users sooner than with the messaging approach.

3.2 Results for all datasets

In this part we analyze all the data obtained during the 3 months the experiment lasted, with the objective to show that the strategy based on the object store solves some of the issues previously described and yields better performance. For example the new approach does not produce the bottlenecks by the head-of-line blockings seen in some cases with the messaging brokers, and improves system scalability. As we can see in Table 3, during the 3 months our experiment lasted both systems were running in parallel, analyzing 26367 datasets containing more than 60 billion events, stored in the worldwide grid in more than 10 million files, and summing up over 17 PB of data. These input data were eventually indexed by 587967 producers. We will compare several metrics, showing the performance capabilities of both systems.

Table 3 Complete results of the experiment comparing messaging and object store approaches

Concept	*Messaging*	*Object Store*
Input Data		
Experiment Duration	3 months	
Total Input Datasets	26367	
Total Files	10 million	
Total Events	60 billion (6180434175 events)	
Total Size	17PB	
Producers		
Instances (short-lived)	587967	
Index results	12311330	
Events indexed	70549949057	
Time spent	3 months (361k CPU hours)	
Transport method	994796792 messages (text)	587967 objects (binary)
Index size	10 TB	2.2 TB (Compressed)
Consumers		
Instances (long-lived)	6	6
Files received (Duplicated)	12311330(2379720)	8663430(0)
Events received	70549949057	53845347540
Receiving method	994796792 messages (text)	538442 objects (binary)
Data received	10 TB	2 TB
Time spent	16728.6 hours	1005.38 hours
Reading rate (kB/s)	173.6 kB/s	571 kB/s
Throughput (events/s)	1171 events/s	14877 events/s
Output data (HDFS)	12311330 HDFS files / 17.7 TB	26367 HDFS files / 14.3 TB
Writing rate (KB/s)	282 kB/s	3950 kB/s
Output Index Data		
Validation Method	Check and remove duplicates	Not needed
Traversal Time (typical)	>10h (94% of datasets)	<10h (85% of datasets)
Consolidated Data (HDFS)	9931610 Files / 14.3 TB / 60 B events	26367 Files / 14.3 TB / 60 B events

Input Data The input data are stored at hundreds of grid sites worldwide, and have to be indexed by the producers that send the results of the indexing process to the central Hadoop instance at CERN. During the time of the experiment, 26367 datasets where indexed, with more than 10 million files that contained more than 60 billion (10^9) events. The total size of the input data to be indexed was in the order of 17 PB of data, with approximately only 10% of the data stored at CERN and 90% of the data stored at grid sites worldwide.

Producers The first phase of the procedure is the indexing phase done by the producers, reading the input data and producing the index data. As said, this step is done only once for both the messaging and object store approaches. The following figures constitute a description of the real workload that is injected to the system in production. We had a total of 587967 producers spawned along the duration of the experiment (3 months) in several grid sites worldwide including CERN, Europe, North America, Asia and Australia.

Figure 5 shows a histogram representing the time spent by producers indexing the input data and producing the index data file. The x-axis represents the duration time (in Hours) of a producer, and the y-axis (logarithmic scale) the number of appearances (frequency) of each case. The indexing time depends on the number of input files and type of data that the producer is handling, but also on the type of machine where the job executes, which in the grid can vary a lot. Most of the producers finish in less than 1 hour (mean is 0.61 hours), and some of them take longer, up to 71 hours. We can count more than 360k hours in total, adding up all the producers that ran during the 3 month experiment.

Figure 6 shows the outcome of the indexing phase, the index data, shown as the number of events indexed per producer. In the x-axis we see the number of events, and in the y-axis (logarithmic scale) the frequency of the producers, the number of appearances of each case. We can see that although the biggest producer job indexes 30 Million events, the most common case is to index 100k events per producer, with a frequency of more than 500k entries. The total number of events indexed is more than the 60 Billion of the input data, because some files are indexed more than once due to some job failures, creating some duplicate data that will be cleaned in a later phase.

In the following figures we can see a characterization and comparison of the quantity of data produced, depending on the approach to encode and convey the data (messaging or object store).

Figure 7 shows a histogram comparing messaging (red) and object store (blue) data encoding methods. This histogram characterizes the produced data size by the producers. In the x-axis we see the size (in MegaBytes) of produced and sent data, and the y-axis represents the frequency or number of occurrences, in logarithmic scale. Here we can see the results of the continuous evaluation during all the time of the experiment (3 months). As we have said, one producer analyzes one or more files of the input data and

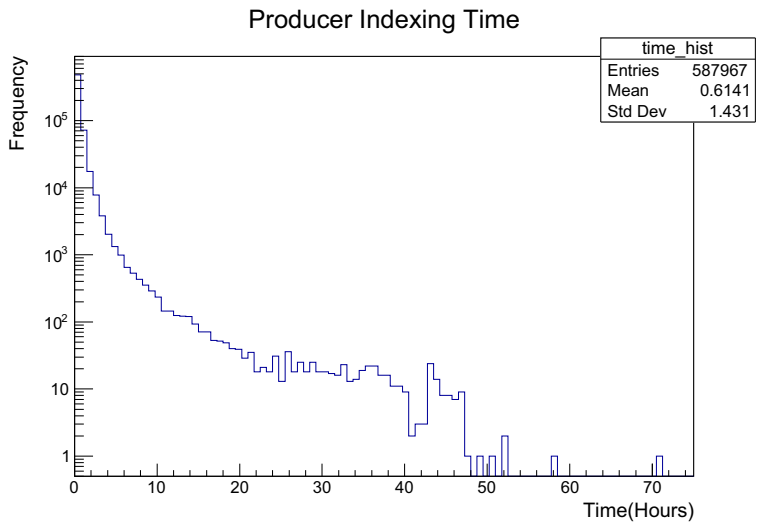

Fig. 5 Producer Indexing Time

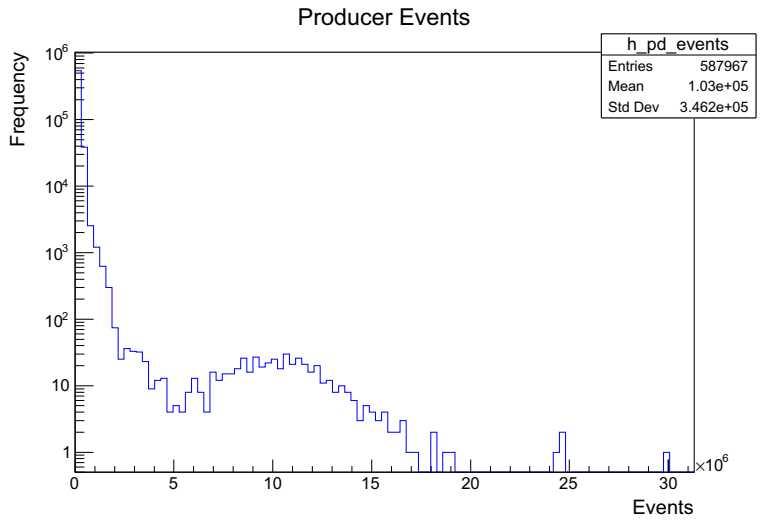

Fig. 6 Number of events indexed per producer

generates index data for the messaging and object store systems. Due to some grid jobs failing and restarting producers, we will have index data for 12 million entries (compared with the 10 million input data files). For messaging (red line) every input file results in one small index file, so in the figure we can see the frequency of the 12 million produced index data sent through the messaging system. We can see that most of the index data is small, with more that 10 million files indexed with producing less than 8 MB of index data per input file (and 99% of the producers send less that 12 MB per indexed file). In fact the mean of the transmitted data per file is 0.8 MB, and only some are bigger with up to 64 MB. The total sum of the transmitted data with the messaging system is 10.5 TB, or around 102 GB/day during the experiment. For object store (in blue), every producer analyzes its input files and stores the index data in a single object in the object store, so we can see that we have much less entries. The data produced was stored in 587867 objects in the object store, with a mean object size of 3.6 MB. Around 90% of the objects are less than 8 MB, but we have some bigger objects up to 670 MB. As a conclusion, we can see that the workload is concentrated in much fewer, but bigger index files (objects in this case). Using the object store

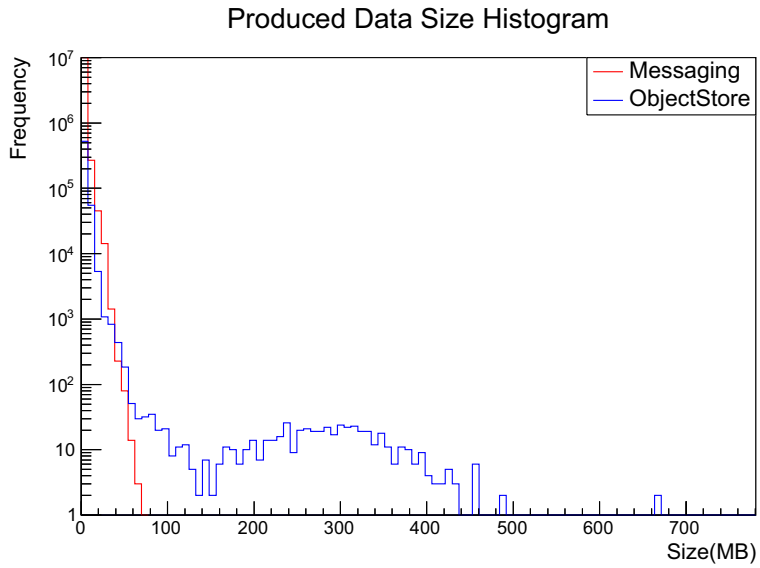

Fig. 7 Comparison of produced index data (messaging vs. object store)

we created 587867 index files(objects), and with the messaging approach we created 12 million index files. The object store approach allows bigger index files (3.6 MB mean index data file, versus 0.8 MB mean index data file with the messaging approach), because we don't have the restrictions imposed by the messaging system and can group information from several input files (additionally compressing the information). The index files sent with the messaging system have to be divided into smaller 10 kB messages, which means millions of messages sent (994,796,792 messages in total). The production of fewer files of bigger size in turns affects the usage of the resources, avoiding the traffic peaks detected in the messaging systems.

Figure 8 shows the amount of data produced per hour for messaging (red) and object store (blue). We see in x-axis the number of elapsed hours since the experiment start, and the y-axis represents the data in MegaBytes. We see the amount of data created by each approach during the 2270 hours the experiment lasted, with a slow start at the beginning and with production peaks at some points. With the messaging approach (red), we can see the peak at hour 1880, that corresponds to almost 60 GB of index data produced during that hour (sent with around 6 million messages). With the object store (blue) we can see a peak with 14.25 GB of created index data (with 3538 objects). It is clearly seen that the amount of data created and sent with the messaging approach is much larger than with the object store approach.

In the following Fig. 9 we see the same information, but accumulating the quantity of data produced until a particular hour since the beginning of the experiment.

We can see in red the total sum of the transmitted data with the messaging system, which is in total 10 TB, or around 102 GB/day during the experiment. In blue, we see the produced data stored in the object store, summing up to a total of 2.2 TB produced data, or a rate of 22.2 GB/day during the time the experiment lasted. Compared with the messaging approach, this means 4.5 times less data sent through the wire. This is also due to the high compression factor that we can achieve with the binary format, that can reach roughly a compression factor 10. By achieving to send the same information with less transmitted data with the object store approach, we can improve the performance of the overall system compared with the previous messaging approach. This will allow us to scale the system in the future ATLAS runs when the higher production rates will create much more events, files and data in total.

Consumers. The differences in consuming approaches can be seen in the following figures, where we break down produced versus consumed data depending on the method, messaging or object store.

Figure 10 shows the number of produced versus consumed messages along the time of the experiment. The red line shows the produced messages, and

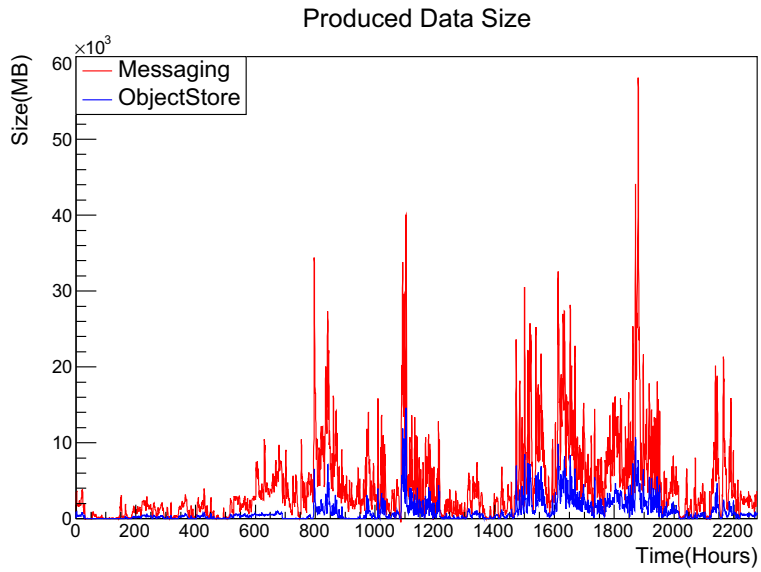

Fig. 8 Produced data(messaging vs. object store) during the duration of the experiment

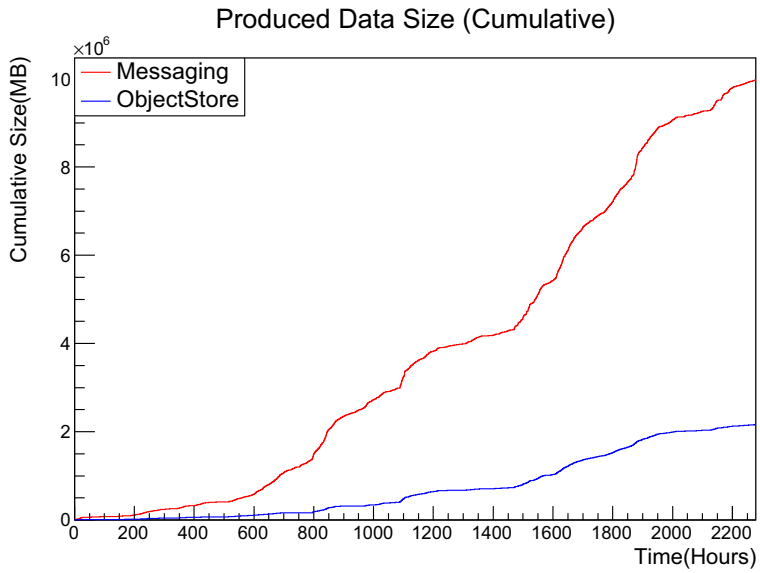

Fig. 9 Cumulative Produced data(messaging vs. object store) during the duration of the experiment

the green line the consumed ones, reaching the 1000 million messages in total, that corresponds to the 10 TB of index data. By design, all messages produced have to be consumed from the messaging broker queues in a constant and ordered manner. In particular all the messages corresponding to a particular index file have to reach together a consumer to be able to reconstruct the original index data file. This is achieved with transactions and messaging groups, but imposes some burdens on the messaging brokers which in some cases can saturate and produce delays. This is what we see in Fig. 10. At the beginning we can see that the consumption of the messages is in line of the production, but in some cases we can see the red line not being followed by the green one, meaning slow consumption of messages. In particular in the time periods where the hour $x \in [1087, 1162]$, $[1189, 1254]$, $[1611, 1748]$ and $[1879, 1912]$, brokers became the bottleneck because of a head-of-line blocking handing the messages to the consumers, and the produced messages were temporarily stored in a backlog queue. In this example there are no lost messages, because all the messages are eventually consumed (both lines produced in red and consumed

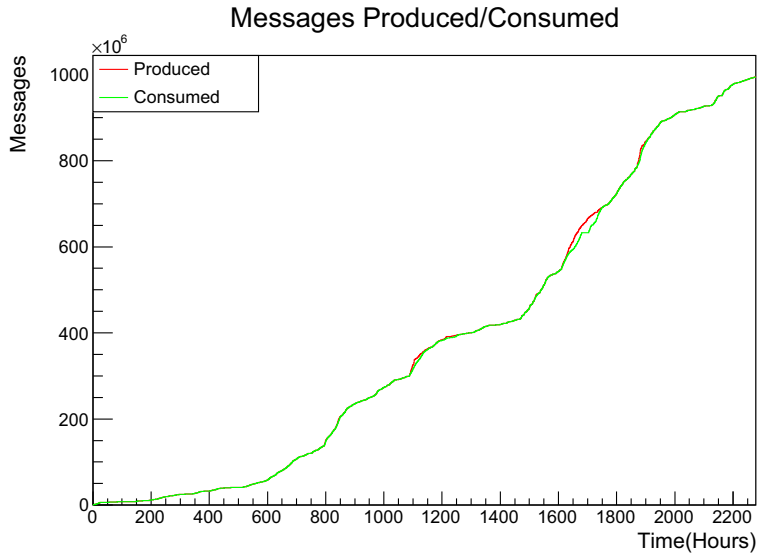

Fig. 10 Produced versus consumed messages

in green converge). But in an extreme situation the brokers at some point might reject more incoming messages if memory is exhausted. Besides slow consumption problems caused by this blockings, there are other issues that make the messaging approach more prone to errors. Problems in the backend HDFS filesystem might delay the consumption of messages, with the consequent creation of a message backlog in the brokers. The system is designed to deal with this kind of situations, but again some messages could be rejected if memory is exhausted. In addition, after a backlog, the consumption is slowed down due to the head-of-line blockings. Also a problem dealing with one message out of all that compose a particular index file, can invalidate the whole index file. This can not be recovered, and the file should be indexed and sent again by a producer.

Figure 11 shows the number of produced versus consumed objects. In red line we can see the produced objects, reaching 587967 total objects. As opposed to the messaging approach, the objects are not inserted into a queue and they have not to be consumed in real time, neither in order. In addition, some produced objects that contain completely duplicate information have not to be consumed at all, which was not possible with the previous messaging approach. This is what we see in this figure, with the green line representing the consumed objects, and reaching 538442 objects consumed in total, meaning that 49525 produced objects with duplicate information which are not accessed at all during consumption. The consumed objects sum up a total of 2 TB of index data. Compared with the original 2.2 TB produced index data data in the object store, this represents 200 GB (10% of the total) of redundant information that does not need to be retrieved. Overall, with the object store approach we need to consume 5 times less amount of data that was consumed with the messaging approach. With these improvements we can reduce the amount of data that is needed to be retrieved by the consumers, spending less time and resources to do this phase of the process. In addition it is an improvement because only valid data is written into HDFS. With the previous messaging approach, an extra validation step is needed to check and remove duplicate data. The index data is not segmented like in the messaging approach, and therefore we don't see blockings or slowed down consumption rates when using the object store. Also the object store approach is more consistent and fault tolerant. In case of temporary faults of the backend HDFS system, the consumers might temporarily stop consuming from the object store. In this situation, an object store is designed to maintain a much larger quantity of (temporary, as in our use case) data than messaging brokers. In addition, when restarting the consumption there are no blockings that might slow down the consumption rates, as in the messaging approach. A problem when dealing and consuming an object, can be solved by reading this object again, as it is placed in the object store

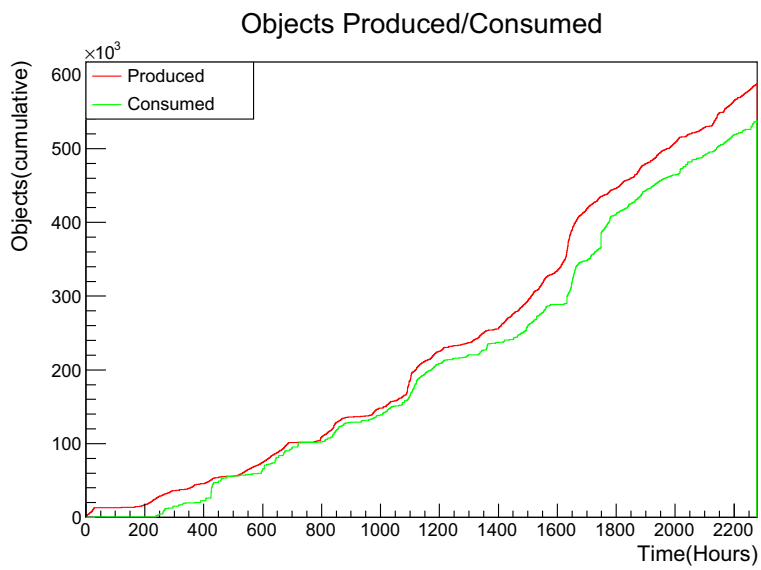

Fig. 11 Produced versus consumed objects

until explicitly deleted. This kind of situation was not possible with the messaging approach.

Validation Other kind of metrics that we can analyze to compare both messaging and object store approaches, is related with the time needed to index and receive valid data. For the final user, the data only make sense when the complete dataset is available. In that sense we define the *traversal time* metric for a particular data or dataset, as the period of time since the first producer starts analyzing data until the produced index data is received and validated by the consumers. This is the main performance metric that the user perceives of the system.

Figure 12 shows a histogram of the traversal time for the same datasets, with both messaging and object store approaches. On the x-axis we see the traversal time in hours, and on the y-axis the relative frequency or number of dataset occurrences normalized to 1 (in logarithmic scale). In blue we can see the traversal time for datasets with the object store path, and we can observe that most of the datasets (85%) are available in less than 10 hours. The traversal mean time for a dataset is $\bar{x} = 12$ hours, with $\text{Var}[x] = 47$. We have some datasets (in the order of tens) consuming more time, and one spurious dataset that reached just over 1000 hours.

In comparison in red we have the traversal time for the same datasets with the messaging path, and we see that only around 6% finish within the first 10 hours. We have much more variance with with $\text{Var}[x] = 247$. We have a non negligible amount of datasets that take a considerable amount of time to be validated, increasing the total traversal time for this datasets up to 1000 hours.

Overall, we see that all datasets flowing through the object store path have a smaller traversal time than the datasets flowing through the messaging path, which means less time until they are available to the final users. We see latency peaks with the messaging approach due to head-of-line blockings that are not present in the object store approach.

Finally, we present the consumers throughput to retrieve data and make them available for final users. This metric is shown in terms of the number of events processed per second, which is independent of the data encoding method.

Figure 13 shows the consumers throughput, in terms of processed events per seconds. On the x-axis we see the number of processed events per second, and on the y-axis the relative frequency of this consumers. In red we see the messaging consumers, with a mean of around 1000 events processed per second, and with a maximum of 5000 events processed per second. In blue we see that the object store consumers get a mean of 15000 events processed per second, and a maximum of 28000 events processed per second. We can see that with the object store approach we get much more performance processing events than with the previous messaging approach, with an

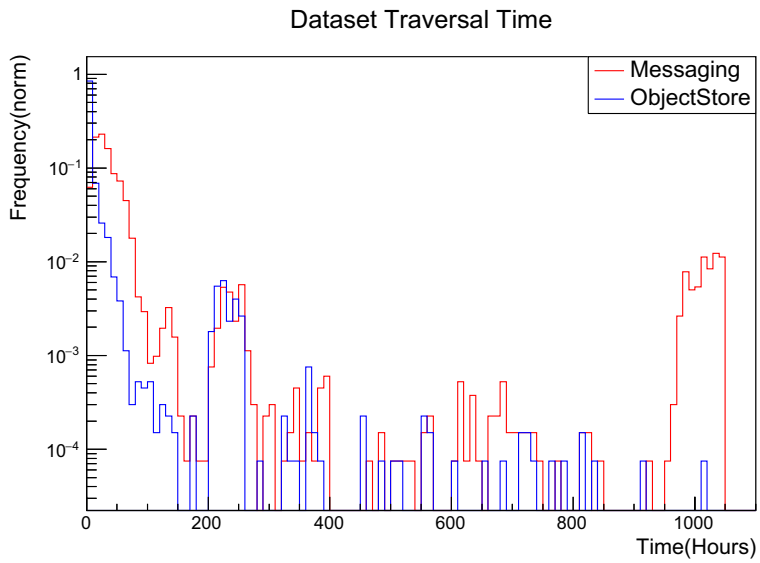

Fig. 12 Comparison of typical traversal times for complete datasets for the messaging and object store approaches

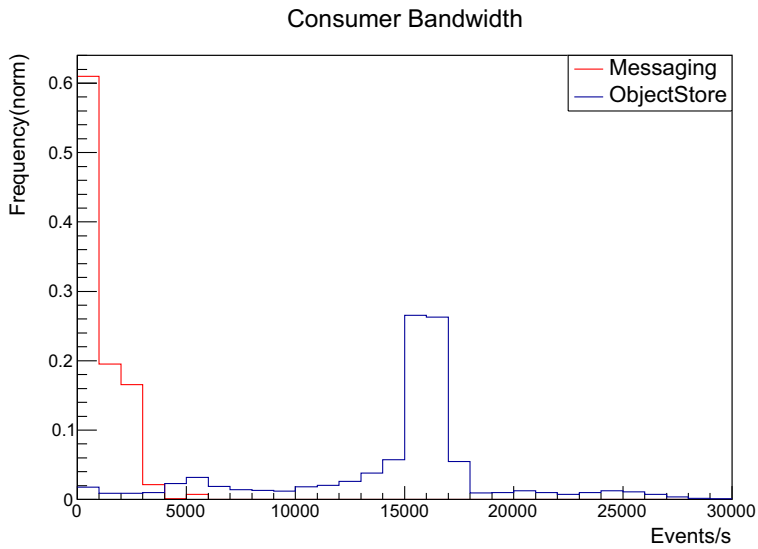

Fig. 13 Comparison of consumer throughput in events processed per second (messaging vs. object store)

improvement by a factor 15. With the blockings detected in the messaging approach, we could not scale the system effectively, since adding new messaging consumers would not result in better throughput. On the other hand, the absence of blockings in the object store approach allows us to scale up by simply adding new consumers.

4 Conclusions

We have presented a new pull-model approach for the distributed data collection of the ATLAS EventIndex project, based on an object store as a shared storage and with dynamic data selection.

In order to compare it experimentally with the previous messaging push-model implementation, we have run both approaches in parallel in our large grid production infrastructure during three months, indexing more than 60 billion events. The results show that the new approach improves the results in several ways. With the new model we can avoid payload segmentation, and store all the information from a producer in a single object. Doing so we can use different data encodings and compression, improving a factor 4.5 in the amount of total produced and conveyed data. We don't see blockings during consumption with the new object store approach, and the workload distribution is improved, making better scalability possible. During our experiments we have seen an improvement of 15 times in the throughput of the consumers, compared with the previous messaging approach. With the pull model we can also select which data to consume dynamically, avoiding duplicate information that might be as much as 10% of the produced data.

We have also improved the way the information is stored in Hadoop, reducing the number of HDFS needed files to only 1 per dataset, meaning a factor 300 improvement with our data. The reduction of complexity and the resource usage, and better performance of the distributed data collection, has improved the experience for final users, that have seen reduced the traversal time, or latency, of the datasets until the indexed data is available.

Overall the results show that the new approach can efficiently support large-scale data collection for big data environments, like the next runs of the ATLAS experiment at CERN.

Acknowledgements This work has been partially supported by MICINN in Spain under grants FPA2016-75141-C2-1-R, PID2109-104301RB-C21 and TIN2015-66972-C5-5-R, which include FEDER funds from the European Union. This work was done as part of the databases applications research and development programme of the ATLAS Collaboration, and we thank the collaboration for its support and cooperation.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. ATLAS Collaboration: The ATLAS experiment at the CERN Large Hadron Collider. J. Instrum. **3**(08), S08003 (2008)
2. Barberis, D., Cárdenas Zárate, S.E., Cranshaw, J., Favareto, A., Fernández Casaní, A., Gallas, E.J., Glasman, C., González De La Hoz, S., Hřivnáč, J., Malon, D., Prokoshin, F., Salt Cairols, J., Sánchez, J., Többicke, R., Yuan, R.: The ATLAS EventIndex: Architecture, design choices, deployment and first operation experience. J. Phys.: Conf. Ser. **664**(4), 042003 (2015)
3. Barberis, D., Cranshaw, J., Favareto, A., Fernández Casaní, A., Gallas, E., González de la Hoz, S., Hřivnáč, J., Malon, D., Nowak, M., Prokoshin, F., Salt, J., Sánchez Martínez, J., Többicke, R., Yuan, R.: The ATLAS EventIndex: Full chain deployment and first operation. Nuclear and Particle Physics Proceedings **273-275**, 913–918 (2016)
4. White, T.: Hadoop: The definitive guide. O'Reilly Media, Inc. (2012)
5. Sánchez, J., Casaní, A.F., de la Hoz, S.G.: Distributed data collection for the ATLAS EventIndex. J. Phys.: Conf. Ser. **664**(4), 042046 (2015)
6. Salomoni, D., Campos, I., Gaido, L., de Lucas, J.M., Solagna, P., Gomes, J., Matyska, L., Fuhrman, P., Hardt, M., Donvito, G., et al: Indigo-datacloud: A platform to facilitate seamless access to e-infrastructures. J Grid Computing **16**(3), 381–408 (2018). https://doi.org/10.1007/s10723-018-9453-3
7. Krašovec, B., Filipčič, A.: Enhancing the grid with cloud computing. J Grid Computing **17**(1), 119–135 (2019)
8. Hagras, T., Atef, A., Mahdy, Y.B.: Greening duplication-based dependent-tasks scheduling on heterogeneous large-scale computing platforms. J. Grid Comput. **19**(1), 13 (2021)
9. Activemq, http://activemq.apache.org/
10. Fernandez Casani, A., Sanchez, J., Gonzalez de la Hoz, S., Orduña, J.M.: Designing Alternative Transport Methods for the Distributed Data Collection of ATLAS EventIndex Project. http://cds.cern.ch/record/2235644/files/ATL-SOFT-SLIDE-2016-869.pdf (2016)
11. Karol, M., Hluchyj, M., Morgan, S.: Input versus output queueing on a space-division packet switch. IEEE Transactions on Communications **35**(12), 1347–1356 (1987). https://doi.org/10.1109/tcom.1987.1096719
12. Mesnier, M., Ganger, G.R., Riedel, E.: Object-based storage. IEEE Commun. Mag. **41**(8), 84–90 (2003). https://doi.org/10.1109/MCOM.2003.1222722
13. Rabbitmq, http://www.rabbitmq.com/
14. Apache flume, http://flume.apache.org/
15. Logstash data processing pipeline, https://www.elastic.co/products/logstash
16. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of NetDB, pp 1–7 (2011)
17. Dobbelaere, P., Esmaili, K.S.: Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, ACM, New York, NY, USA, pp 227–238 (2017). https://doi.org/10.1145/3093742.3093908
18. Bird, I.G.: Lhc computing (wlcg): Past, present, and future. Grid and Cloud Computing: Concepts and Practical Applications **192**, 1 (2016)
19. Garonne, V., Vigne, R., Stewart, G., Barisits, M., Lassnig, M., Serfon, C., Goossens, L., Nairz, A., Collaboration, A., et al: Rucio–the next generation of large scale distributed system for atlas data management. In: Journal of Physics: Conference Series, IOP Publishing, 513, pp 042021 (2014)
20. Calafiura, P., De, K., Guan, W., Maeno, T., Nilsson, P., Oleynik, D., Panitkin, S., Tsulaia, V., Gemmeren, P.V., Wenaus, T.: The atlas event service: A new approach to event processing. J. Phys.: Conf. Ser. **664**(6), 062065 (2015)
21. Amazon simple storage service developer guide, http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf
22. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, USENIX Association, Berkeley, CA, USA, pp 307–320 (2006)
23. Ceph, http://ceph.com/
24. Amazon s3, cloud computing storage for files, images, videos. http://aws.amazon.com
25. Google protocol buffers: Google's data interchange format. https://developers.google.com/protocol-buffers/