EPJ Web of Conferences **214**, 04057 (2019) https://doi.org/10.1051/epjconf/201921404057
CHEP 2018

A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage

Zbigniew Baranowski[1,*], *Luca* Canali[1,**], *Alvaro* Fernandez Casani[2,***], *Elizabeth J* Gallas[3,****], *Carlos* Garcia Montoro[2,†], *Santiago* González de la Hoz[2,‡], *Julius* Hrivnac[4,§], *Fedor* Prokoshin[5,¶], *Grigori* Rybkine[4,‖], *Jose* Salt[2,**], *Javier* Sanchez[2,††], and *Dario* Barberis[6,‡‡] on behalf of the ATLAS Collaboration

[1]CERN, Geneva, Switzerland
[2]Insitut de Fisica Corpuscular, Valencia Spain
[3]University of Oxford, Denys Wilkinson Bldg, Keble Rd, Oxford OX1 3RH, United Kingdom
[4]LAL, Université Paris-Sud and CNRS/IN2P3, Orsay, France
[5]Universidad Tecnica Federico Santa Maria, Chile
[6]Università di Genova and INFN, Genova, Italy

Abstract. The ATLAS EventIndex has been in operation since the beginning of LHC Run 2 in 2015. Like all software projects, its components have been constantly evolving and improving in performance. The main data store in Hadoop, based on MapFiles and HBase, can work for the rest of Run 2 but new solutions are explored for the future. Kudu offers an interesting environment, with a mixture of BigData and relational database features, which look promising at the design level. This environment is used to build a prototype to measure the scaling capabilities as functions of data input rates, total data volumes and data query and retrieval rates. In this proceedings we report on the selected data schemas and on the current performance measurements with the Kudu prototype.

1 Introduction

The ATLAS EventIndex [1] is a catalogue of all real and simulated events produced by the experiment [2] at all processing stages. The system contains hundreds of billions of event records (180 billion records as of June 2018), each consisting of approximately 1000 bytes. The goal of the ATLAS EventIndex is to allow fast and efficient selection of events of interest,

*e-mail: zbigniew.baranowski@cern.ch
**e-mail: luca.canali@cern.ch
***e-mail: Alvaro.Fernandez@ific.uv.es
****e-mail: elizabeth.gallas@physics.ox.ac.uk
†e-mail: carlos.garcia@ific.uv.es
‡e-mail: sgonzale@ific.uv.es
§e-mail: Julius.Hrivnac@cern.ch
¶e-mail: Fedor.Prokoshin@cern.ch
‖e-mail: Grigori.Rybkine@cern.ch
**e-mail: Jose.Salt@ific.uv.es
††e-mail: Javier.Sanchez@ific.uv.es
‡‡e-mail: Dario.Barberis@cern.ch

based on various criteria, and provide references that point to those events in millions of files scattered in a world-wide distributed computing system.

2 Motivation for the project evolution

The current EventIndex was designed in 2012-2013 using the best BigData technology available at that time (Hadoop), implemented in 2014 using MapFiles and HBase, and in operation since 2015 with satisfactory results [3]. However, the use cases evolved in the meantime and have been extended from event picking and production completeness checks to trigger overlap studies, duplicate event detection and derivation streams (offline triggers) overlaps. At the same time, the implementation of fast data querying based on a traditional relational database involves only a subset of information and is available for real events [4]. Also the event rate increased steadily throughout Run 2.
In the meantime BigData technologies advanced and now we have the choice between many different products and options. Studies of new data formats and/or new storage technologies [5] concluded that Kudu is the most promising technology for the next few years. Hence this prototype.

3 Apache Kudu

Apache Kudu is a next-generation scalable and distributed table-based storage designed for HTAP systems – Hybrid Transactional and Analytical Processing [6]. Unlike most of the data formats available for the Hadoop Distributed File System (HDFS) [1], Kudu provides indexing and columnar data organization natively – this is to establish a good compromise between random data lookups and analytics performance. The data organization in shared tables with named columns, types and a primary index makes Kudu very attractive for systems with relational data models that need to scale out. Apache Kudu is supported by top open-source frameworks for parallel data processing and computation including Apache Spark, Apache Impala, Apache Hive, MapReduce and others.

4 Concept of the new ATLAS EventIndex platform

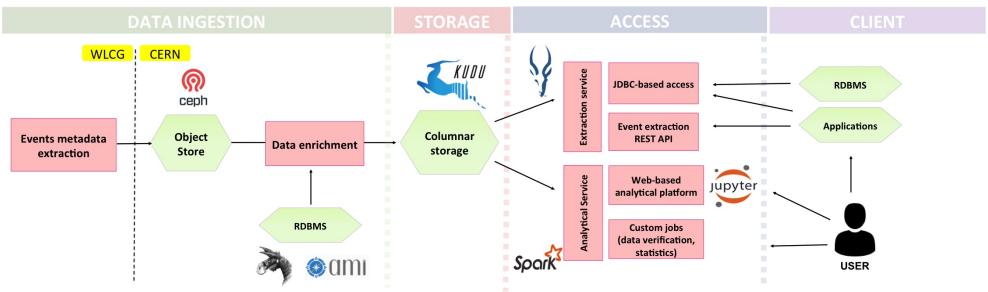

Figure 1. The new system architecture

Bringing on-board Kudu, a storage supported by many computing frameworks, offers to the ATLAS EventIndex a possibility to have a modular architecture and increases its flexibility in

[1] http://hadoop.apache.org [accessed 2018-07]

further evolution. Any module responsible today for one of the functions, the data ingestion layer, data access layer or data analytic layer could be replaced with a modern one, without significant effort.
In the new architecture presented in Figure 1, we foresee to keep the data ingestion/production part without significant modification. It has been already modernized by introducing a staging area implemented with an object store layer [7] (based on Ceph storage) staging all recently indexed event data from WLCG [2] jobs. The only work foreseen in this area is to build a data pipeline between the object store and the Kudu storage. Such a pipeline will need to enrich the data with necessary information regarding trigger state, event provenance etc. from the AMI [8] and Rucio [9] metadata systems.

The main modernization touches the data access layer that has to be redesigned in order to fully profit from the functionality and performance offered by Apache Kudu. Notably, the MapReduce framework has to be replaced with low-latency SQL-based frameworks like Apache Impala [3]. Impala is well integrated with Kudu and allows to start event lookups queries within milliseconds. Moreover, the addition of an SQL interface to the ATLAS EventIndex data will open a possibility to integrate the project with external tools and systems over the JDBC protocol. So eventually currently available web front-ends implemented for a relational back-end can be reconnected to the Kudu-based storage.
As more analytic use cases have been added to the system recently, Apache Spark [4], a modern and highly scalable and feature-rich data processing engine, would be an obvious choice for the implementation of such use cases. With Spark, a lot of complex computations can be expressed imperatively in a high level programming language (Scala, Python or Java), in a much easier way than the same logic with SQL. Finally, thanks to the available integration of Spark with online Jupyter Notebooks (CERN's SWAN project [5]), we can empower the end-users to write their own analyses and execute them on ATLAS EventIndex data.

5 Schema design in Apache Kudu

When designing a schema in Apache Kudu we had to make decisions on a few aspects that ultimately have a great impact on a data access time, scalability and usability of the data stored in Kudu tables. This includes the number of tables and relations between them, choosing appropriate primary keys and partitioning strategies for each of the tables, and finally, data types, encodings and compression algorithms for each of the columns in the tables.
In this section, we will describe the schema for Kudu tables that we prototyped to store the ATLAS EventIndex data and criteria that led us to make certain design choices.

5.1 Technology constraints

Along with a certain number of Apache Kudu limitations [6], there are a few that impose certain schema design constraints. Notably, 1) each table has only one index and it is built on a primary key; 2) a table partition is a unit of table scan parallelism; 3) each Kudu cluster node can only host 5000 partitions; 4) the partition range has to be known during creation and cannot be modified; 5) the cardinality of key-hash-based sub-partitioning is static and cannot be changed for each partition individually; 6) a table cannot have more than 256 columns.

[2] http://wlcg.web.cern.ch [accessed 2018-07]
[3] https://impala.apache.org [accessed 2018-07]
[4] https://spark.apache.org [accessed 2018-07]
[5] https://swan.web.cern.ch [accessed 2018-07]
[6] https://kudu.apache.org/docs/known_issues.html [accessed 2018-07]

5.2 Partitioning strategy

Knowing the constraints mentioned in the previous sections we had to invoke certain techniques to go around them. This mainly applies to the non-primary key data access. Such an access path requires a scan of all table columns that are part of a query predicate (filtering clause) or projection (selection clause). This quite expensive task, when comparing to the index-based access, can be executed by multiple parallel processes on the Kudu cluster and effectively reduce greatly the response time. The more partitions (including sub-partitions) the input table has, the bigger parallelism can be achieved - this is due to the limitation 2) mentioned in the previous section. For a single range partition, we have decided to have 64 hash-based sub-partitions. This will give at least 64 parallel table scanners when running queries that do not invoke primary key-based access.
Furthermore, in order to reduce the work done by the scanners, the number of rows to be visited by the scanning operation can be narrowed/pruned to those stored in certain partitions. This can happen if a partitioning key is present in a filter predicate of a query. In theory the 'runnumber' column from ATLAS EventIndex data is the best candidate for a table partition key, as it exists in all the queries and it has very high selectivity, as only a few partitions will be chosen for scanning. However, since the amount of data per run number is skewed, we would have too many almost empty partitions and sub-partitions too. With respect to the limitation 3) we would quickly reach the maximum number of supported partitions. We could group more runs into a single partition in order to keep all partitions with a unified size; however, due to the limitation 4) grouping cannot be done dynamically. To overcome the problem of statically defined partition keys we have to introduce an artificial partition key for event data. Its value is an incremental number, and it has to be controlled by the data ingestion supervisor. Once a certain partition is big enough, a new one should be created with a key incremented by one. The relation between partition keys and datasets has to be kept in a metadata table and looked up by each query.

5.3 Tables

Following a process of multiple iterations, we have concluded that the following list of tables should be sufficient to store the system data in an efficient way and provide the required functionality:

- *DATASETS* – metadata table, grouping events into datasets. This table assigns a unique id for each dataset, maps each dataset to a partition, and provides basic statistics about each dataset, notably the number of events, import time, how much time it took to load it and corresponding statistics copied from the AMI system for consistency verification. The full schema of the table can be found in the Table 1. Since this table is expected to not grow significantly in size, it does not have to be partitioned.

- *EVENTS* – contains metadata about each event. Most of the data access use cases will query this table to obtain information about the events, such as: trigger chain, GUID of a file with actual event data, and provenance of an event - what files it originated from. The full schema of the table can be found in Table 2. Because this table is subject to grow at a high rate, it has to be partitioned according to rules described in section 5.2. On top of it, we have partitioned it with a hash function into 64 buckets using an event number column, in order to achieve a high degree of parallelism on evenly distributed data.

Table 1. *Datasets* table schema

column name	column type	encoding	compression	primary key
runnumber	int	BIT_SHUFFLE	LZ4	X
project	string	DICT_ENCODING	SNAPPY	X
streamname	string	DICT_ENCODING	SNAPPY	X
prodstep	string	DICT_ENCODING	SNAPPY	X
datatype	string	DICT_ENCODING	SNAPPY	X
version	string	DICT_ENCODING	SNAPPY	X
dspid	int	BIT_SHUFFLE	LZ4	
rgid	int	BIT_SHUFFLE	LZ4	
insert_start	timestamp	BIT_SHUFFLE	LZ4	
insert_end	timestamp	BIT_SHUFFLE	LZ4	
backup_start	timestamp	BIT_SHUFFLE	LZ4	
backup_end	timestamp	BIT_SHUFFLE	LZ4	
validated	timestamp	BIT_SHUFFLE	LZ4	
count_events	bigint	BIT_SHUFFLE	LZ4	
uniq_dupl_events	bigint	BIT_SHUFFLE	LZ4	
num_duplicates	bigint	BIT_SHUFFLE	LZ4	
tigger_counted	int	BIT_SHUFFLE	LZ4	
ds_overlaps	int	BIT_SHUFFLE	LZ4	
ami_count	bigint	BIT_SHUFFLE	LZ4	
ami_raw_count	bigint	BIT_SHUFFLE	LZ4	
ami_date	timestamp	BIT_SHUFFLE	LZ4	
ami_upd_date	timestamp	BIT_SHUFFLE	LZ4	
ami_state	string	DICT_ENCODING	SNAPPY	
inconctainer	int	BIT_SHUFFLE	LZ4	
state	string	DICT_ENCODING	SNAPPY	
smk	int	BIT_SHUFFLE	LZ4	

Table 2. *Events* table schema

column name	column type	encoding	compression	primary key
dspid	int	BIT_SHUFFLE	LZ4	X
eventnumber	bigint	BIT_SHUFFLE	LZ4	X
rgid	int	BIT_SHUFFLE	LZ4	X
hltpsk	int	BIT_SHUFFLE	LZ4	
l1psk	int	BIT_SHUFFLE	LZ4	
lumiblocknr	int	BIT_SHUFFLE	LZ4	
bunchid	int	BIT_SHUFFLE	LZ4	
eventtime	int	BIT_SHUFFLE	LZ4	
eventtimens	int	BIT_SHUFFLE	LZ4	
lvl1id	bigint	BIT_SHUFFLE	LZ4	
l1trigmask	string	DICT_ENCODING	SNAPPY	
l1trigchainstav	string	DICT_ENCODING	SNAPPY	
l1trigchainstap	string	DICT_ENCODING	SNAPPY	
l1trigchainstbp	string	DICT_ENCODING	SNAPPY	
eftrigmask	string	DICT_ENCODING	SNAPPY	
eftrigchainsph	string	DICT_ENCODING	SNAPPY	
eftrigchainspt	string	DICT_ENCODING	SNAPPY	
eftrigchainsrs	string	DICT_ENCODING	SNAPPY	
dbraw	string	DICT_ENCODING	SNAPPY	
tkraw	string	DICT_ENCODING	SNAPPY	
dbesd	string	DICT_ENCODING	SNAPPY	
tkesd	string	DICT_ENCODING	SNAPPY	
dbaod	string	DICT_ENCODING	SNAPPY	
tkaod	string	DICT_ENCODING	SNAPPY	
db	string	DICT_ENCODING	SNAPPY	
tk	string	DICT_ENCODING	SNAPPY	

- *DUPLICATEDEVENTS* – this table contains all duplicated events that could not be stored in the *events* table due to the primary key violation. It has exactly the same schema as *events* table except one extra column *insertion_time* which is a part of the primary key.

- *MCEVENTS* – unlike the *events* table, this one stores only simulated events. It has a similar schema to the *events*, however some columns related to event provenance information have been dropped since they are not applicable to simulated events. The partitioning strategy should be the same as in case of the *events* table.

- *EVENTSOVERLAP* – the purpose of this table is to store the results of datasets overlap computations - how many events two given datasets are sharing. Therefore it has a basic schema, containing two primary key columns defining datasets id being compared, and a big integer counter column. This table is expected to grow slowly, so no partitioning is needed.

- *TRIGGERCOUNT* – similarly to *eventsoverlap*, it should store the results of computations on trigger masks. This table has just 4 columns. Three - *dataset_id*, *trig1* and *trig2* are making the primary key. The remaining column is just a big integer counter, representing the

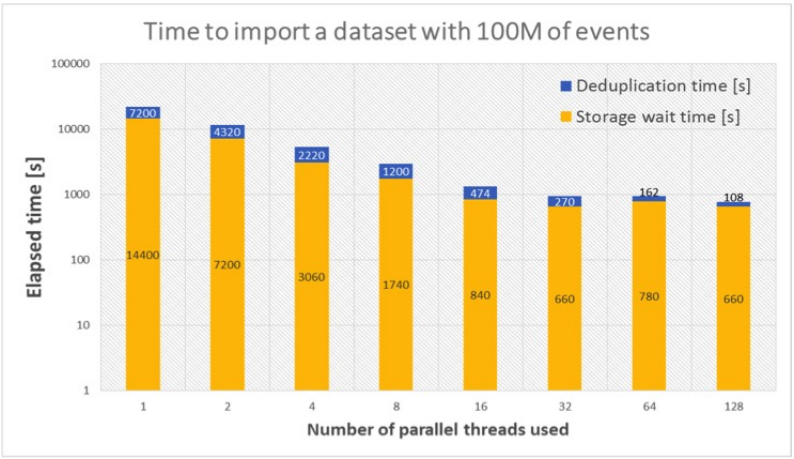

Figure 2. Data ingestion speed measured on representative data set

number of occurrences of the two given triggers within a dataset. The partitioning strategy is the same as in the case of the *events* table, however hashing is dropped, as parallel, non-primary key data access is not required for this table.

Notably, we used bit shuffle encoding for numbers and timestamps and dictionary encoding for strings. Regarding the selection of compression algorithm for columns in the tables, we use the following rule: LZ4 for all the columns with bit shuffle encoding and snappy for all the columns with dictionary encoding. All this made a single event record occupying 200 bytes on disk on average (reduced by a factor of 5 in comparison to the original data volume).

5.4 Performance of the prototype

In order to understand the efficiency of the designed schema, a series of performance tests have been concluded with the most important system use cases: event data ingestion, event lookup and analytics on trigger data.

5.4.1 Hardware and software used

All the performance tests have been performed on a cluster composed of 12 physical machines, each equipped with 2 CPUs with 8 physical cores with clock speed 2.60 GHz, 64 GB of RAM and 48 SAS drives with 4 TB each. Apache Kudu was compiled from version 1.7.0 on the SLC6 Linux distribution (equivalent to RedHat 6). The software was configured to use at maximum 10 GB of RAM and 10 cores for maintenance tasks.

5.4.2 Data ingestion

Data loading tests were performed with Apache Spark 2.2.1 using real data from the current production system. Before loading a dataset to Apache Kudu, duplicated events are filtered out and stored in a dedicated table. The performance tests have been concluded on multiple datasets with random size and a varaying number of parallel ingestion processes. Figure 2 shows the results of the set of tests performed on a representative dataset with 100M events

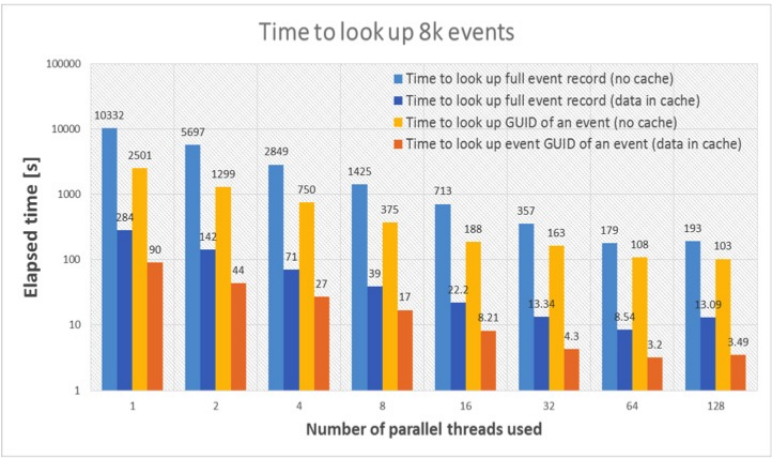

Figure 3. Data access to a representative data set

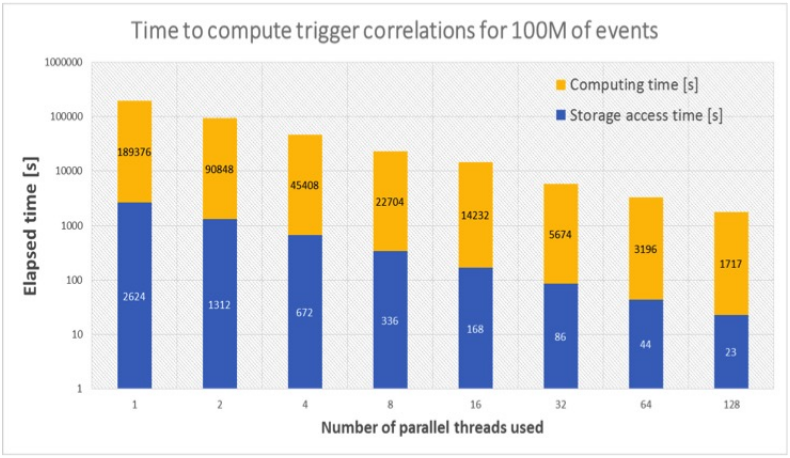

Figure 4. Data analytic speed measured on representative data set

to demonstrate the scalability of data ingestion. The measured average writing speed was 5 kHz per thread, and the maximum overall writing speed was 120 kHz. The obtained result is about 10 times more than what is needed today by the project.

5.4.3 Data access

In Figure 3 we present the time to look up eight thousand random event records (full record or just a GUID attribute) from Apache Kudu with a simple client written in Python. The results for each type of a lookup were grouped into two cases; a pessimistic one (no cache used) and an optimistic one (all data were looked up from the Kudu cache). The average event lookup time below 1 s and the ability to handle more then 400 requests per second fully satisfies the future system needs.

5.4.4 Analytics

Data analytics tests were performed with Apache Spark 2.2.1 reading ATLAS EventIndex records from data stored in Apache Kudu. Figure 4 shows the execution times for the test

case, where we count the occurrences of all combinations of trigger bit pairs within a dataset of 100M events. The trigger count computation on a Spark cluster takes the majority of the wall time (52 hours), with data retrieval from Kudu being just a small fraction of it (45 minutes, or < 2%). A single scanner thread could deliver 40k records per second.

6 Full data flow tests

We want to test a schema that is flexible to evolve, and check the performance that we can obtain with the data ingestion procedure. The testbed setup that we have used for these tests is located at IFIC in Valencia, and is currently composed of 5 machines with $2 \times$ *Xeon* E5-2690 CPU, and 256 GB of memory each. The software installation included Kudu release 1.7, Impala 2.11 for accessing the data, and Spark 1.6 from the Cloudera distribution release cdh5.14.2. The current Kudu configuration uses 1 of these machines as master, and the other 4 as tablet servers. There are 8×6TB data hard disk per machine configured as one big Raid 10 disk to store tablets, summing up 22TB per machine to a total of 88TB of storage. The write ahead log is located in the extra NVMe SSD of 1.5 TB per machine to improve the performance of the write operations.

6.1 Trigger encoding tests

Trigger data is one the biggest components in size for every stored event. It is important to find a good encoding and compression schema for its components, as it may reduce the total size of the EventIndex store and can dramatically increase the performance of the complete system. We performed some tests to check several encoding and compression algorithms available within the Kudu instance. The input data is the trigger information for Level1 and the High Level Trigger, and for both sets we check JSON and binary encoding with several options and compression schemas. For example the columnar representation in Kudu allows to store a binary trigger in one or more columns to allow selections on individual triggers. We can benefit from Impala bit wise operations to access individual bits (representing one trigger) in a single performant operation, with the limitation that the operand, and so the column size, is 64 bit long. In this case we can represent the 1536 bits of Level 1 trigger (L1) as 24 columns of 64 bits each, and the 12288 bits of the High Level Trigger (HLT) with 192 columns. We can benefit from accessing only the columns that contain the specific triggers that we want to access (projection push-down), omitting unnecessary fields when doing table scans. We also wanted to test a JSON representation of the trigger, as it provides versatility to operations and some databases also currently offer direct access to JSON fields in a performant manner. We have just one string column for representing the Level 1 in JSON (L1mask) and another for the High Level Trigger (HLTmask).

Table 3 shows the results of the encoding tests, representing the number of bytes an event occupies in the storage using the corresponding encoding and codification and compression schemas. In the first set of results we can see the results of binary encoding for L1 and HLT. The best ratio of bytes per event is obtained with bit shuffle encoding, which rearranges the bits for achieving better compression ratios relying on neighbouring elements. In addition, for our trigger use case there are usually many bits that are 0, i.e. not signal. In this case we do not explicitly set them to 0, but leave the field as null as much as possible. Then we apply a compression algorithm, with LZ4 being the best option for this kind of bit shuffle encoding. In this case we obtain the best results: 60 and 15 bytes/event for L1 and HLT respectively.

In the second set of results we can check the JSON encoding of the trigger data. In the first column we try the Dictionary encoding, best suitable when we have a small number of entries that is highly repeated. Without compression this is clearly the worst selection.

Table 3. Column total size in Bytes/event

binary	BIT_SHUFFLE LZ4 (0 as null)	PLAIN_ENCODING LZ4 (0 as null)	PLAIN_ENCODING LZ4 (0 as 0)
L1	59.96	65.17	69.28
HLT	14.97	26.96	45.16
JSON	DICTIONARY NO_COMPRESSION	PLAIN_ENCODING LZ4 (O as text)	
L1Mask	697.55	150.93	
HLTMask	680.86	45.47	

6.2 Ingestion tests

In this section we review the ingestion tests done with a typical Consumer adapted to write into Kudu. We selected the datasets from Tier-0 recorded during May 2018 as input for our test experiment, and we tested 4 different partition schemas independently. Figure 5 shows the number of ingested events per second for a single consumer on the x-axis, with the 4 schemas in different colors. The schema 'base-t1' in black, represents 8 partitions based on the hash of the event number of the events, and 8 partitions based on the run number. In our case datasets from the same month have very similar run numbers, so partitioning on the run number does not affect the result. The schema 'base-t2' in red follows the same approach, but with the difference that the primary key of the table ends with <...,runnumber,eventnumber>, reversing the fields of 'base-t1'. Most of the datasets are ingested with a performance of 5k events per second. Epoch based schemas in blue and green use a range partition based on an epoch that can be set per dataset, instead of a runnumber, in order to distribute it more evenly. In this case we obtain in the range from 5k to 6k events per second. There are spurious better performances and lower performances for the smaller datasets, but we can claim that the ingestion mean rate is roughly 5k events per second. In Figure 6 we see a breakdown of the time that a Consumer uses doing the ingestion, by stage type. Basically for most of the schemas it spends 1% of the time waiting for data from the Object Store, then 4% of the time doing parse and data conversion from the input data, and then the insertion phase into Kudu client buffers is roughly 23% of the time, with the last flush phase taking the bulk of the time (72%). Only the first bar that represents inserting duplicates shows a difference, doubling the insert time phase.

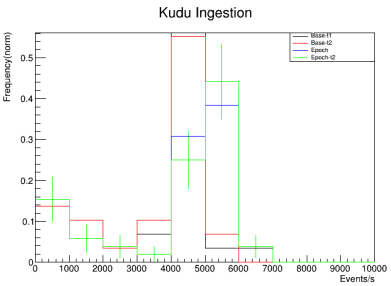

Figure 5. Event ingestion in Apache Kudu

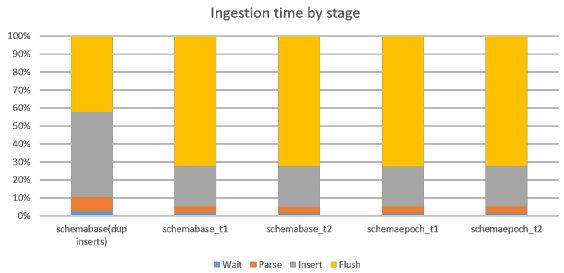

Figure 6. Consumer Ingestion stages

7 Summary

The prototype and the performed tests have proven that the ATLAS EventIndex can profit from migrating the core storage to Apache Kudu in many aspects. Apache Kudu allows

maintaining a hybrid system in a single piece of software. Notably, it enables features like streaming data ingestion, fast data access by index, and columnar storage for analytics. An eventual migration to a new back-end brings some obvious costs to the project, as it requires the re-implementation of most of the current components. However, it should bring a lot of simplification to the code as the vast majority of the logic needed so far is already built-in in existing frameworks that support Kudu or in Kudu itself. The scalable data scan performance in combination with modern data processors (Spark, Impala) opens the system to new use cases on a field of data exploration and analytics (like counting trigger correlations).

Out of many assets of Apache Kudu, there are also few downsides of it. The biggest one appears to be its maturity and production readiness - the product is available since two years, but although we have been running it without any stability problems for months without major problems there are no companies officially using Apache Kudu in production.

To conclude, feature-wise Apache Kudu appears to be very strong, fulfilling the ATLAS EventIndex project needs for the LHC Run 3. However, from a strategic point of view it is still an unknown area. Is it there to stay for years? Regardless of the technology used in the future by the Atlas EventIndex project, the work done on the evolution of the system in many aspects is universal and can be re-used on the majority of data store back-ends that support data relationality.

References

[1] Barberis D et al. (2014) The ATLAS EventIndex: an event catalogue for experiments collecting large amounts of data, J. Phys.: Conf. Ser. 513 042002

[2] ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, JINST 3 (2008) S08003

[3] Barberis D et al. (2015) The ATLAS EventIndex: architecture, design choices, deployment and first operation experience, J. Phys. Conf. Ser. 664 042003

[4] Gallas E et al., (2017), an Oracle-based EventIndex for ATLAS, J. Phys.: Conf. Ser. 898 042033

[5] Z. Baranowski et al (2017), A study of data representation in Hadoop to optimize data storage and search performance for the ATLAS EventIndex, J. Phys.: Conf. Ser. 898 062020

[6] Lipcon T et al., (2015), Kudu: Storage for fast analytics on fast data.

[7] Daniel van der Ster, Arne Wiebalck (2014), Building an organic block storage service at CERN with Ceph ,J. Phys. Conf. Ser. 513 042047

[8] J Fulachier et al (2017), ATLAS Metadata Interface (AMI), a generic metadata framework, J. Phys.: Conf. Ser. 898 062001

[9] V Garonne et al (2014), Rucio – The next generation of large scale distributed system for ATLAS Data Management, J. Phys.: Conf. Ser. 513 042021