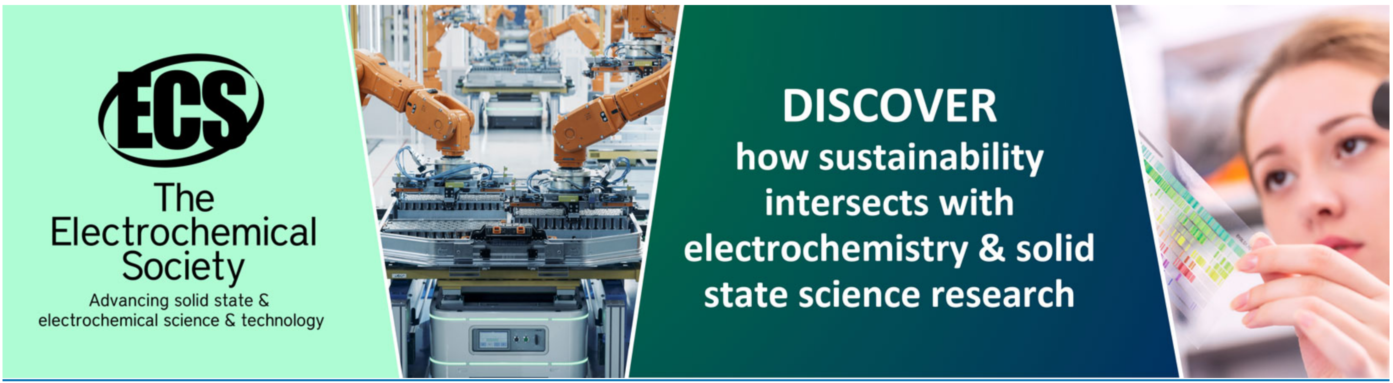
Journal of Physics: Conference Series

PAPER • OPEN ACCESS

A study of data representation in Hadoop to optimize data storage and search performance for the ATLAS EventIndex

To cite this article: Z. Baranowski *et al* 2017 *J. Phys.: Conf. Ser.* **898** 062020

View the article online for updates and enhancements.


ECS
The Electrochemical Society
Advancing solid state & electrochemical science & technology
DISCOVER
how sustainability intersects with electrochemistry & solid state science research

A study of data representation in Hadoop to optimize data storage and search performance for the ATLAS EventIndex

Z. Baranowski[1] L. Canali[1] R. Toebbicke[1] J. Hrivnac[2] D. Barberis[3]

[1] CERN, Geneva, Switzerland
[2] LAL, Université Paris-Sud and CNRS/IN2P3, Orsay, France
[3] Università di Genova and INFN, Genova, Italy

Email: *zbigniew.baranowski@cern.ch*

Abstract. This paper reports on the activities aimed at improving the architecture and performance of the ATLAS EventIndex implementation in Hadoop. The EventIndex contains tens of billions of event records, each of which consists of ~100 bytes, all having the same probability to be searched or counted. Data formats represent one important area for optimizing the performance and storage footprint of applications based on Hadoop. This work reports on the production usage and on tests using several data formats including Map Files, Apache Parquet, Avro, and various compression algorithms. The query engine plays also a critical role in the architecture. We report also on the use of HBase for the EventIndex, focussing on the optimizations performed in production and on the scalability tests. Additional engines that have been tested include Cloudera Impala, in particular for its SQL interface, and the optimizations for data warehouse workloads and reports.

1. The ATLAS EventIndex project

The ATLAS EventIndex [1] is a catalogue of all real and simulated data produced by the ATLAS experiment [2], one of seven particle detectors constructed for the CERN Large Hadron Collider [3]. The EventIndex was designed in 2012-2013 and implemented in 2014; the first data (all LHC Run 1 data collected in 2009-2013) were loaded at the beginning of 2015.

1.1. System requirements

The ATLAS EventIndex system has to scale to the order of several 10^{10} events (the number of events expected for LHC Run 2 between 2015 and 2018), be flexible in its schemas to accommodate a variety of quantities to be stored that could change in the future, use established and possibly open-source technologies and be “easy” to develop, deploy and operate.

The main use cases that were identified for this service are [4] event picking, trigger checks, event skimming and production consistency checks.

1.2. Current architecture

The ATLAS EventIndex collects the indexing data from distributed computer centres and stores them in a central storage at CERN. The system is divided into few functional packages that implement data acquisition, storage, access and monitoring.

The storage component called *Core Storage* is one of the critical parts of the system as it integrates all other packages by consolidating the data and making them available to be accessed by users via the *Query Server* component. Thus it is important that it is robust and delivers the required performance for both data ingestion and data access. Apache Hadoop [5] was chosen as the main backend technology for storing and accessing data. It met all criteria from the project requirements (in section 1.1) and, differently from other shared storage technologies like relational databases, in various tests at CERN [6] has proven that it is horizontally scalable.

The rest of this paper elaborates on the internal design of the Core Storage and potential improvements that can bring the use of alternative approaches available in the Hadoop ecosystem for storing and accessing the data.

2. Core Storage – implementation, usage and bottlenecks

In order to understand how the core layer of the ATLAS EventIndex can profit from adopting recent technologies, it is important to explain some key implementation aspects of this component.

2.1. Storage implementation

The Core Storage package is responsible for implementing persistent storage based on Apache Hadoop and implementing data accessing interfaces. It consists of two components: *Catalog* and *HDFS namespace*. The first one is an inventory of all imported datasets and their internal schema. All catalogue data are stored in an HBase database running on the same Hadoop cluster as the EventIndex data. The EventIndex data are physically stored on the Hadoop Distributed Files system in a format called MapFile [7]. A MapFile is a union of two tSequence files [8] that have two functions. The first one holds raw data and the second one contains key-based index to the data stored in the first file.

2.2. Event Index record content

Each indexed event is stored in a MapFile format as a separate record that in average is 1.5 kB long and has 56 attributes encoded in various types. Most of them are arrays of characters, few are integers and floating point numbers. The main attributes are:

- *Event identification*: run number (integer), event number (long), trigger stream (string), event format (string) and processing version (string).
- *Trigger information*: the list of trigger chains passed by the given event (string).
- *References of the event*: the GUIDs (Global Unique IDentifiers) (string).

In the third quarter of 2016 there are 6x10^9 records stored in HDFS that occupy tens of Terabytes.

2.3. Data access paths

There are two typical access paths that satisfy all the use cases. The first one is *event picking* – lookup for a random event by identification attributes. It is the main use case of the ATLAS EventIndex. Until the end of 2015, this access was implemented by using Catalogue-based data pruning and lookup for relevant records in a MapFile by a built-in index. The second access path is *data scanning* – full scan of a population of events in order to perform event skimming or trigger-based selection. For this type of access path, a MapReduce job is used in order to perform distributed and scalable data filtering.

2.4. Limitation of the Core Storage implementation

During a review of the ATLAS EventIndex project in late 2015, a few limitations have been identified in the Core Storage implementation:

- Data ingestion into MapFile format is complex, as it requires sorting datasets by key values before storing them physically in HDFS. The average measured ingestion speed into MapFile format was 6.4 kHz per dataset. A dataset corresponds to a data-taking run, typically up to 10^8 events.
- Due to the extra effort (mentioned above) needed when loading data into MapFiles, a number of staging areas with duplicated data are created and maintained.
- EventIndex data in MapFiles and metadata in HBase are separated and served by different components. This means that each data access operation has to have an extra cost (latency) of performing a metadata lookups in order to obtain the data location and the data schema.

- Random data lookup of MapFiles is performed on the client side. This can potentially cause a performance problem when the network connectivity between HDFS and the client is poor or when a single client machine performs multiple requests in parallel like in the case of Query Server. Typical single event lookup speed when using MapFiles is around 4 seconds.

Most of the limitations identified during the review were related to the usage of the MapFile file format as a container for the data. For this reason, a new initiative of evaluating alternative possibilities of storing data in the Hadoop ecosystem was started. The main goal was to understand if Core Storage could significantly profit from using a different format for the data representation.

3. Evaluation of alternative modern storage approaches for Core Storage

This chapter describes a performance comparison of some popular data formats and storage engines available in the Hadoop ecosystem to evaluate space efficiency, ingestion performance, analytic scans and random data lookup. This should help in understanding how (and when) each of the evaluated technologies can improve handling of the ATLAS EventIndex big data workloads.

During the evaluation, the same ATLAS EventIndex data sets have been stored on the same Hadoop cluster using different storage techniques and compression algorithms supported by them (Snappy, GZip or BZip2; the compression settings were tuned natively by each of the storage technology).

3.1. Hardware and storage configuration

The data access and ingestion tests were performed on a cluster composed of 14 physical machines, each equipped with 2 CPUs with 8 physical cores with clock speed 2.60GHz, 64 GB of RAM and 48 SAS drives, 4TB each. Hadoop was installed from Cloudera Data Hub (CDH) distribution version 5.7.0, which includes, *Hadoop core 2.6.0*, *Impala 2.5.0*, *Hive 1.1.0*, *HBase 1.2.0* (configured JVM heap size for region servers = 30 GB) and *Kudu 1.0* (configured memory limit = 30 GB). Apache Impala (incubating) was used as a data ingestion and data access framework in all the conducted tests presented later in this report.

3.2. Evaluated formats and technologies

With respect to recent trends on the market and evaluations done with various storage techniques in the past at CERN [9], four candidate technologies for storing the data in the Hadoop ecosystem have been chosen.

3.2.1. Apache Avro [10] is a data serialization standard for compact binary format widely used for storing persistent data in HDFS as well as for communication protocols. One of the advantages of using Avro is lightweight and fast data serialization and deserialization, which can deliver very good ingestion performance.

Even though it does not have any internal index (like in the case of MapFiles), the HDFS directory-based partitioning technique can be applied to quickly navigate to the collections of interest when fast random data access is needed. In the test a tuple of *runnumber*, *project* and *streamname* was used as a partitioning key. This allowed obtaining good balance between the number of partitions (few thousands) and an average partitions size (hundreds of megabytes). Two supported by Apache Avro algorithms were used in the tests: Snappy and DEFLATE.

3.2.2. Apache Parquet [11] is a column-oriented data serialization standard for efficient data analytics. Additional optimizations include encodings (Run Length Encoding, Dictionary, Bit packing), and the compression applied on series of values from the same columns that gives very good compaction ratios. When storing data in HDFS in Parquet format, the same partitioning strategy was used as in the Avro case. Two Apache Parquet supported algorithms have been used to compressed data: Snappy and GZip.

3.2.3. Apache HBase [12] is a scalable and distributed NoSQL database on HDFS for storing key-value pairs. Keys are indexed, which typically provides very quick access to the records. When storing ATLAS EventIndex data into HBase each event attribute was stored in a separate cell, and the row key was composed as a concatenation of an event identification attributes (*runnumber*, *eventnumber*, *project*, *streamname*, *datatype* and *version*). Additionally, encoding of the row key was enabled in order to reduce the size of HBase blocks (without this, each row would have the length of 8KB). HBase supports 3 algorithms for data blocks compression – Snappy, GZip, LZO. Two of them: Snappy and GZip were applied in the tests.

3.2.4. Apache Kudu [13] is a new scalable and distributed table-based storage. Kudu provides indexing and columnar data organization to achieve a good compromise between ingestion speed and analytics performance. In the evaluation all literal types were set to be stored with a dictionary encoding and numeric types with bit shuffle encoding. Additionally, a combination of range and hash partitioning was introduced, by using the first column (*runnumber*) of the primary key (composed of the same event attributes like in HBase case) as a partitioning key. Two out of three supported by Apache Kudu compression algorithms were used for storing the data: Snappy and zlib.

3.3. Measurement results
Despite the effort made to obtain as precise results as possible, they should not be treated as universal and fundamental benchmarks of the tested technologies. There are too many variables that could influence the tests and make them more case specific, like the chosen test cases, the data model used, the hardware specification and configuration, and the software stack used for data processing.

3.3.1. Space utilization
Measuring the average record size after storing the same data sets (1.1 billion of records) using different techniques and compression algorithms allows estimating what would be the expected volume of production data when migrated to the chosen format and the space savings associated with that.

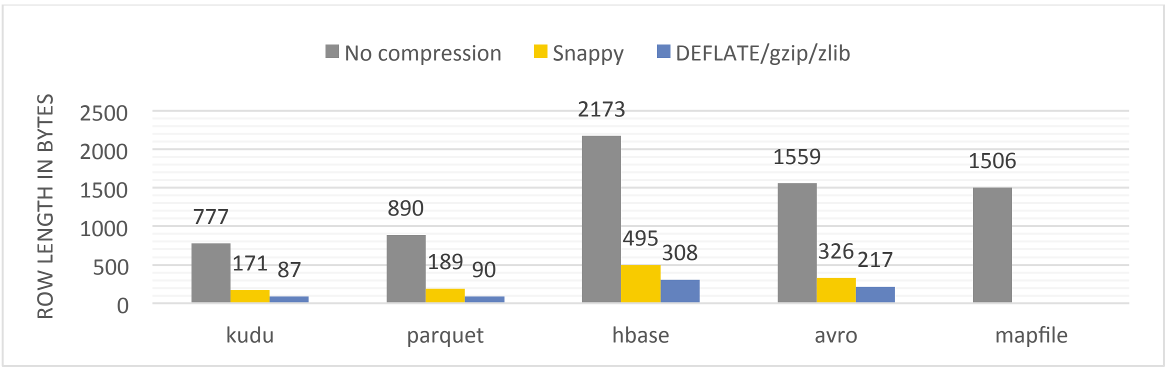


Figure 1: Average row length in bytes for each tested format and compression type.

According to the measured results (Figure 1), data encoded with Kudu and Parquet delivered the best compaction ratios. Using compression algorithms like Snappy or GZip can further reduce the volume significantly – by a factor 10 comparing to the original data set encoding with MapFiles.

HBase, due to the way it stores the data, is a less space efficient solution. Although compressing the HBase blocks gives quite good ratios, however, it is still far away from those obtain with Kudu and Parquet.

On the other hand, Apache Avro delivers similar results in terms of space occupancy like other HDFS row stores e.g. MapFiles.

3.3.2. Ingestion speed

Measuring records ingestion speed into a single data partition should reflect the performance of writing to the ATLAS EventIndex Core Storage system that can be expected when using different storage techniques. The results of this test are presented on Figure 2.

In general, it is difficult to make a valid performance comparison between writing data to files and writing data to a storage engine. However, because Apache Impala performs writing into a single HDFS directory (Hive partition) serially, the results obtained for HDFS formats and HBase or Kudu can be directly compared for single data partition ingestion efficiency.

Writing to HDFS files encoded with Avro or Parquet delivered much better results (at least by a factor 5) than storage engines like HBase and Kudu. Since Avro has the most lightweight encoder, it achieved the best ingestion performance. At the other end of the spectrum, HBase in this test was very slow (worse than Kudu). This most likely was caused by the length of the row key (6 concatenated columns), that in average was around 60 bytes. HBase has to encode a key for each of the columns in a row separately, which for long records (with many columns) can be suboptimal.

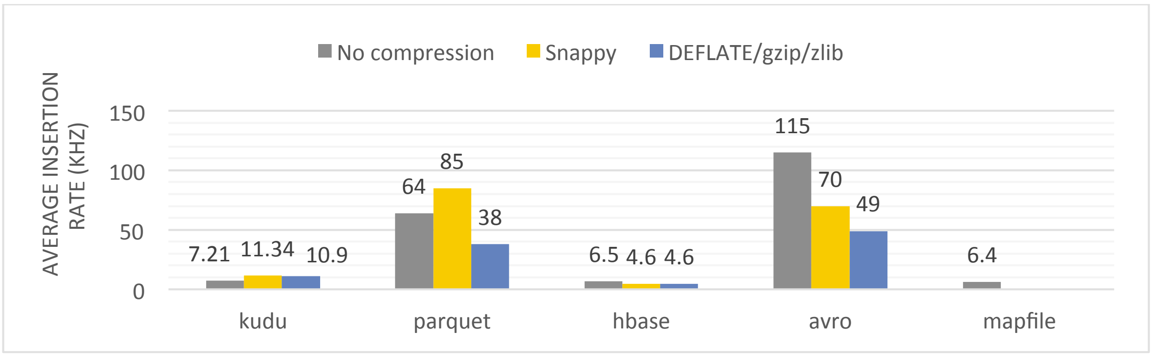


Figure 2: Average ingestion speed in kHz into a single data partition for each tested data format.

3.3.3. Random data lookup

Retrieving a non-key attribute from a record by providing a record identifier (a compound key) is the main use case of the EventIndex (see 1.1). With respect to that, a list of runnumber-eventnumber pairs was used in order to retrieve their corresponding GUID.

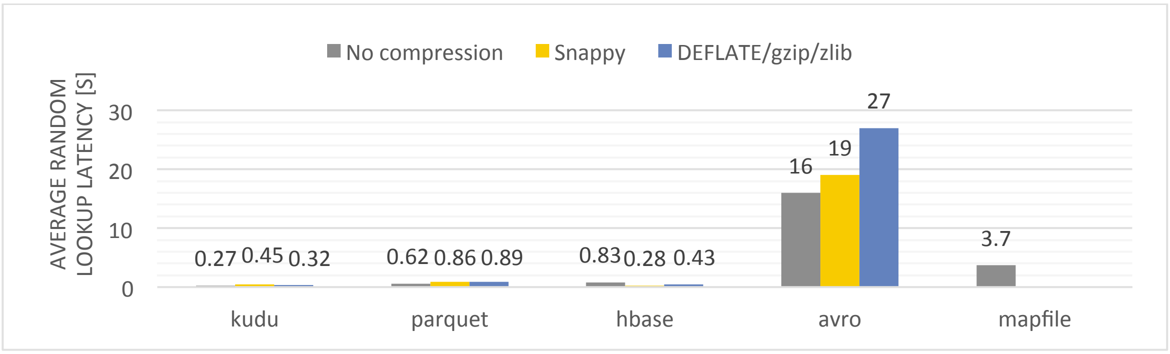


Figure 3: Average random record lookup latency [in seconds] per data format.

According to the measured results (Figure 3), when accessing data by a record key, Kudu and HBase were the fastest ones, because of the usage of built-in indexing. Values on the plot were measured with cold caches. Using Apache Impala for random lookup test is suboptimal for Kudu and HBase as a significant amount of time is spent to set up a query (planning, code generation etc.) before it really gets executed – typically this takes about 200 ms. Therefore for low latency data access it is advised to skip Impala and use dedicated APIs; we tried also this approach and results for Kudu and HBase were similar – with cold cache <200 ms and with warmed up cache <80 ms.

In opposite to Kudu and HBase, retrieving data from an individual record stored in Avro format can only be done in a brute force scan of an entire data partition (as mentioned in 3.2.1 – the data are partitioned by part of a record key, so all the partitions that do not contain such key can be filtered out). An average partition is sized in GB, thus getting the desired record takes seconds (depending on I/O throughput) and uses a significant amount of the cluster resources. This ultimately reduces the number of concurrent queries that can be executed at a full speed on a cluster.

The same problem applies to Parquet; however, the columnar nature of the format allows performing partition scans relatively fast. Thanks to column projection and column predicate push down, a scan input set is ultimately reduced from GBs to just a few MBs (effectively only 3 columns were scanned out of 56).

3.3.4. Data processing speed

The data scanning test was performed as a simplified use case of event skimming or trigger based selection. This simulates extraction of a trigger chain information from all the events and later count only the ones that met certain condition (substring matching).

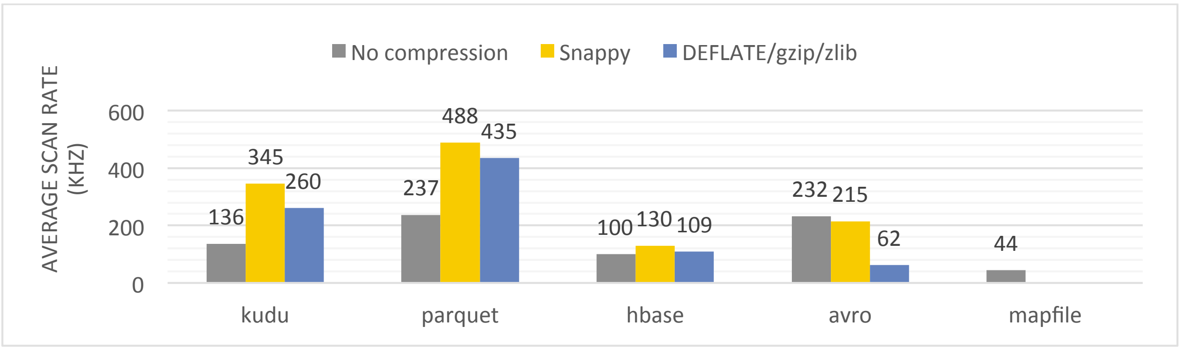


Figure 4: Average scan speed per CPU core [kHz] for each tested format.

The unit of data access parallelization in the case of Parquet and Avro is an HDFS file block – thanks to that it is very easy to evenly distribute processing across all the resources available on a Hadoop cluster. Due to the input set reduction by applying column projection, Parquet in this test has left behind Avro (Figure 4). It was not only the most efficient in terms of per-core processing rates but also the fastest to finish the processing.

In terms of scanning efficiency Kudu (with Snappy compression) was not far from Parquet. It profited from column projection. Scanning data stored in Kudu and HBase might be imbalanced since a single table partition is the unit of a scan parallelization in both cases. Therefore the amount of resource involved in a scan depends on the number of given table partitions, their sizes and their distribution across a cluster. In this test case, it was not possible to use Kudu's native predicate push down feature, as Kudu did not support the used predicate. Additional tests proved that, when supported predicates are in use, Kudu scans could be as fast as those performed on Parquet format.

Before performing the test with HBase the scanned column was separated in a dedicated HBase column family – this improved the scanning efficiency by factor 5. That was still far away from Parquet or Kudu.

3.4. Summary of the evaluation

The performed tests of major data storing techniques with the ATLAS EvenIndex workloads delivered valuable information about the key aspects to be considered when deciding to deploy any of these techniques:

- *Storage efficiency* – with Parquet or Kudu and Snappy compression the total volume of the data can be reduced by a factor 10 comparing to uncompressed simple serialization format.
- *Data ingestion speed* – all tested file based solutions provide faster ingestion rates (between x2 and x10) than specialized storage engines or MapFiles (sorted sequence).
- *Random data access time* – using HBase or Kudu, typical random data lookup speed is below 500ms. With smart HDFS namespace partitioning Parquet could deliver random lookup on a level of a second but consumes more resources.
- *Data analytics* – with Parquet or Kudu it is possible to perform fast and scalable (typically more than 300k records per second per CPU core) data aggregation, filtering and reporting.
- *Support of in-place data mutation* – HBase and Kudu can modify records (schema and values) in-place where it is not possible with data stored directly in HDFS files.

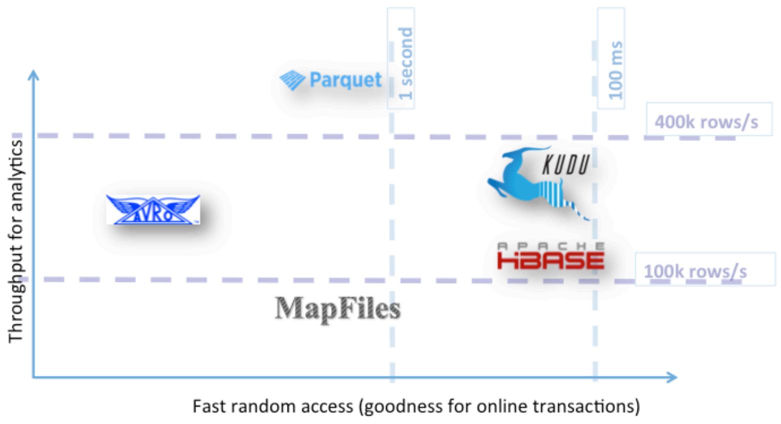


Figure 5: A schematic view of the results of the tests on Hadoop data formats and storage engines. Kudu and Parquet appear as good compromises between random data lookup and scalable data analytics performance.

Apache Avro has proven to be a fast universal encoder for structured data. Due to very efficient serialization and deserialization, this format can guarantee very good performance whenever an access to all the attributes of a record is required at the same time – data transportation, staging areas etc.

On the other hand Apache HBase delivers very good random data access performance and the biggest flexibility in structuring stored data (schema-less tables). The performance of batch processing of HBase data heavily depends on a chosen data model and typically cannot compete on this field with the other tested technologies. Therefore any analytics with HBase data should be performed rather rarely.

Notably, compression algorithms played a significant role not only in reducing the data volume but also in enhancing the performance of data ingestion and data access. In all those fields the Snappy codec delivered the best results for all tested technologies, much better than plain encoding without compression (except in the case of Avro).

4. Hybrid system

Alternatively to a single storage technology implementation, a hybrid system could be considered composed of a raw storage for batch processing (like Parquet) and indexing layer (like HBase) for random access. This would allow to fully profit from technologies specialization/optimization on certain access paths and deliver the best performance. Notably, such approach comes at the price of data duplication, overall complexity of a system architecture and higher maintenance costs.

At the end of 2015 as a follow-up of the initial evaluation of available storage techniques, an attempt for building a hybrid system for the ATLAS EventIndex was conducted in two ways:

- Indexing the most relevant data (event identification and references of the event) in a separate relational system (Oracle) [14]. The assumption here was that this index should be self-contained and does not keep pointers to the complete event records available on HDFS.

- Indexing events by event number and run number in HBase database. In this approach the indexing key resolves to GUID and pointers to the complete records stored on HDFS.

So far both systems have proven to deliver very good events picking performance on a level of tens of milliseconds – two orders of magnitude faster than the original approach when using MapFiles solely. The only concern when running a hybrid approach in both cases is the system size and internal coherence – robust procedures for handling HDFS raw data sets updates and propagating them to indexing databases with low latency have to be maintained and monitored.

5. Conclusions

The study of improving the ATLAS EventIndex Core Storage performance has shown the potential for enhancing the current implementation efficiency in many aspects, like reduction of overall data volume, simplifying ingestion and increasing the performance of data access. Columnar stores like Apache Parquet and Apache Kudu appear to be very good candidates for future data storage systems as they guarantee very good flexibility between fast data ingestion, fast random data lookup and scalable data analytics by keeping the system simplicity (Figure 5). On this field, Kudu appears to be more suited for the ATLAS EventIndex use case because of fast event lookup and simplified ingestion procedures. However, deep evaluation of Apache Kudu disclosed a lack of important functionalities (like security) and maintenance problems that makes Kudu in the currently available version (1.0.1) not fully production ready.

On the other hand, deployment of additional indexing platforms to improve fast data access (HBase and Oracle) provided satisfactory results for the main use cases of the ATLAS EventIndex. This came at a price of extra complexity of the system and extra maintenance effort. However, at the given state of development of the project, it pays off.

In the longer term, there are plans to consolidate the data onto a single platform. With respect to that, Apache Kudu seems to be the best choice. Therefore further monitoring of the technology evolution is foreseen.

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 doi:10.1088/1742-6596/513/4/042002
[2] ATLAS Collaboration 2008 The ATLAS Experiment at the CERN Large Hadron Collider, JINST 3 S08003 doi:10.1088/1748-0221/3/08/S08003
[3] Large Hadroon Collider http://home.cern/topics/large-hadron-collider
[4] Barberis D et al. 2015 The ATLAS EventIndex: architecture, design choices, deployment and first operation experience, J. Phys. Conf. Ser. 664 042003 doi:10.1088/1742-6596/664/4/042003
[5] http://hadoop.apache.org
[6] Baranowski Z, Canali L and Grancher E 2013 Sequential data access with Oracle and Hadoop: a performance comparison, J. Phys. Conf. Ser. 513 042001 doi:10.1088/1742-6596/513/4/042001
[7] https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/MapFile.html
[8] https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/SequenceFile.html
[9] Baranowski Z et al. 2015 Scale out databases for CERN use cases, J. Phys. Conf. Ser. 664 042002 doi:10.1088/1742-6596/664/4/042002
[10] https://avro.apache.org
[11] https://parquet.apache.org
[12] https://hbase.apache.org
[13] https://kudu.apache.org
[14] Gallas E et al. 2016 an Oracle-based Event Index for ATLAS, (to appear in J. Phys.: Conf. Ser., Computing in High Energy and Nuclear Physics 2016 International Conference)