Proceedings of the 9th International Conference "Distributed Computing and Grid-technologies in Science and Education" (GRID'2021), Dubna, Russia, July 5-9, 2021

THE ATLAS EVENTINDEX USING THE HBASE/PHOENIX STORAGE SOLUTION

Elizaveta Cherepanova[1,a], Evgeny Alexandrov[1], Igor Alexandrov[1], Dario Barberis[2,b], Luca Canali[3], Alvaro Fernandez Casani[4], Elizabeth Gallas[5], Carlos Garcia Montoro[4], Santiago Gonzalez de la Hoz[4], Julius Hrivnac[6], Andrei Kazymov[1], Mikhail Mineev[1], Fedor Prokoshin[1], Grigori Rybkin[6], Javier Sanchez[4], José Salt Cairols[4], Miguel Villaplana Perez[4], Aleksandr Yakovlev[1]

on behalf of the ATLAS Software and Computing Activity

[1] Joint Institute for Nuclear Research, Joliot-Curie 6, RU-141980 Dubna, Russia

[2] Università di Genova and INFN, Via Dodecaneso 33, I-16146 Genova, Italy

[3] CERN, CH-1211 Geneva 23, Switzerland

[4] Instituto de Física Corpuscular (IFIC), Univ. de Valencia and CSIC,
C/Catedrático José Beltrán 2, ES-46980 Paterna, Valencia, Spain

[5] University of Oxford, Wellington Square, Oxford OX1 2JD, United Kingdom

[6] IJCLab, Université Paris-Saclay and CNRS/IN2P3, 15 rue Georges Clémenceau,
FR-91405 Orsay, France

E-mail: [a] Elizaveta.Cherepanova@cern.ch, [b] Dario.Barberis@cern.ch

The ATLAS EventIndex provides a global event catalogue and event-level metadata for ATLAS analysis groups and users. The LHC Run 3, starting in 2022, will see increased data-taking and simulation production rates, with which the current infrastructure would still cope but may be stretched to its limits by the end of Run 3. This talk describes the implementation of a new core storage service that will provide at least the same functionality as the current one for increased data ingestion and search rates, and with increasing volumes of stored data. It is based on a set of HBase tables, coupled to Apache Phoenix for data access; in this way we will add to the advantages of a BigData based storage system the possibility of SQL as well as NoSQL data access, which allows the re-use of most of the existing code for metadata integration.

Keywords: Scientific computing, BigData, Hadoop, HBase, Apache Phoenix, EventIndex

1. Introduction

The ATLAS Collaboration [1] developed the EventIndex [2] to store information about the main properties of each real or simulated event and pointers to the files that contain it. The current EventIndex storage implementation reflects the state of the art for BigData storage tools in 2012-2013 when the project started, but many different options have appeared since, even within the Hadoop [3] ecosystem that is used as the main data store. With the increase of data-taking and simulation production rates foreseen for Run 3 (2022-2024) and even more for Run 4 (High-Luminosity LHC, from 2027 onwards), a re-design of the core systems is needed. In order to be safe, a new system should be able to absorb a factor 10 higher event rate than the current one, *i.e.* 100 billion real events and 300 billion simulated events each year.

Investigations on several structured storage formats for the main EventIndex data to replace the Hadoop MapFiles [4] used till now started a few years ago [5]. Initially it looked like Apache Kudu [6] would be a good solution, as it joins BigData storage performance with SQL query capabilities [7]. Unfortunately Kudu did not get a sufficiently large support in the open-source community and CERN decided not to invest hardware and human resources in this technology.

HBase [8] had been evaluated as the main data store at the beginning of the project but was discarded at that time because of performance restrictions. Nowadays instead, it can hold the large amounts of data to be recorded, with a much-improved data ingestion and query performance thanks to the increased parallelisation of all operations. Additional tools like Apache Phoenix [9] can provide SQL access to HBase tables, if the tables are designed appropriately upfront, which can be done in our case.

While updating the core storage system, all other components have to be revised and if necessary updated or replaced. In particular, the Data Collection system and the Supervisor [10] need to be extended to cover the complete data flow through the whole system.

2. Evolution of the Data Collection system

The main motivation for the further evolution of the Data Collection system is the usage of modern storage like HBase/Phoenix, and data processing technologies like Spark [11], as detailed in section 3. It will also allow to simplify all procedures, reducing data duplication and using common job management tools over the stored data. These include data mangling, calculation of duplicates, overlaps, and trigger. In addition, we will maintain the performance according to the production rates, for acceptable data traversal times. The current approach is to keep the Producer-Consumer architecture, as natural for the distributed nature of the ATLAS Grid infrastructure, and a single Supervisor to orchestrate the procedure.

The Producer implementation is currently done in python with a single thread. It will be upgraded to work with the latest data analysis software and external libraries like stomp.py [12], boto [13] and Protocol Buffers [14].

The CERN Object Store [15] will still be used to maintain an intermediate temporary storage, from where the Consumers retrieve the data to the final HBase/Phoenix destination. The use of the EOS store at CERN [16] will remain as a fallback mechanism when the Object Store is not available or accessible from the distributed worker nodes.

The Supervisor will be improved in several areas:

- Ability to control the creation of indexing tasks: The current supervisor discovers indexation tasks sent by separate data production processes directly to the Grid workload management system PanDA [17] when their first job finish. The new supervisor will interact directly with PanDA to allow full control of the indexation process from the very beginning.
- Increased granularity in the indexation process: Sometimes the indexation tasks fail because a few files of a dataset were unavailable. In this case the task is now discarded and the whole dataset is re-

indexed. The new supervisor will be able to create sub-datasets that contain just the files whose indexation has failed, avoiding the re-indexation of the whole task.

- Increased granularity of the data consumption: The current supervisor validates the dataset once it is completely indexed and then signals the consumers to start its consumption. This approach is inefficient for large datasets in case the ingestion process is interrupted due to service problems or input file corruption and a process that may have been running for several hours has to be restarted. The new supervisor will be able to split the information consumption into smaller chunks.

The Consumers can currently write to Phoenix as standalone processes, but in the new approach they will be converted to Spark [11] jobs that can run in the data storage infrastructure and be scaled up when necessary.

The calculation of duplicates, overlaps or other analytic jobs over larger amounts of data will be done offline with Spark jobs over the Hbase/Phoenix stored data.

3. Data structures in HBase/Phoenix

HBase works best for random access, which is perfect for the event picking use case where we want low-latency access to a particular event to get its location information. Use cases where we need information retrieval (trigger info, provenance) for particular events are served by fast HBase gets. In addition, analytic use cases where we need to access a range of event information for one or several datasets (derivation or trigger overlaps calculation), can be solved with range scans on these data. They can be optimized with a careful table and key design that maintain related data close within the storage, reducing access time.

HBase is a column-family grouped key-value store, so we can benefit from dividing the event information into different families according to the data accessed in separated use cases; for example, we can maintain event location, provenance, and trigger information in different families.

Apache Phoenix is a layer over HBase that enables SQL access and provides an easy entry point for users and other applications. Although HBase is a schema-less storage, Apache Phoenix requires a schema and data typing to provide its SQL functionalities.

3.1 Events table

Row keys

Best performance is gained from using row keys. We are going to have several billions of entries and we want single row access and scans to be as efficient as possible, so we need to include the most needed information in the key and leave other information in column families. Pure value-based access is always a full scan, so all the index information should be in the row key for better performance, while maintaining the row key size to its minimum. A representation of the schema of the events table can be seen in Figure 1.

An event record is uniquely identified by its dataset name: *Project.RunNumber.StreamName. prodStep.dataType.AMItag_[tidN]* and its *EventNumber*. To satisfy all use cases we need to access different information:

- Event picking: needs to know the EventNumber and which dataset to get the event from.
- Event selection based on trigger: needs to know the dataset name and selection criteria.
- Derivation overlap: needs to run over all the derivations for datasets having the same dataset name except for the dataType.

Therefore we include this information in the row key chosen as a composite value *dspid.dstypeid.eventno.se*q (16 bytes), where:

- dspid (Integer: 4 bytes) is an identifier for the dataset name, excluding the dataType.
- dstype (Integer: 2 bytes) is an identifier for the dataType.
- eventno (Long: 8 bytes) is the event number.

- seq (Short: 2 bytes) is used to deduplicate event entries when the EventNumber collides.

The *dspid* is generated by the Supervisor during data ingestion. Generating monotonically increasing values is not good for HBase, as it can create hot spots and not distribute the load among regions. This can be solved by reversing the bit order (which has the property to distribute and cover the whole key space from the beginning).

The *dstype* allows scanning for the datasets (having the same dspid) over all the data types, which is the use case of dataset overlaps computation. It is internally computed into dataTypeFormat (5 bits = 32 values) and dataTypeGroup (11 bits = 2 048 values) for optimal usage.

The *seq* is computed as the crc16 [18] value of (GUID:OID1-OID2), where GUID [19] is the identifier of the file containing the event, and OID1-OID2 are the internal pointers within that file. There is a chance of key clashing, but it was estimated to be low enough for our purpose whilst keeping the row key small over other alternatives.

Data families

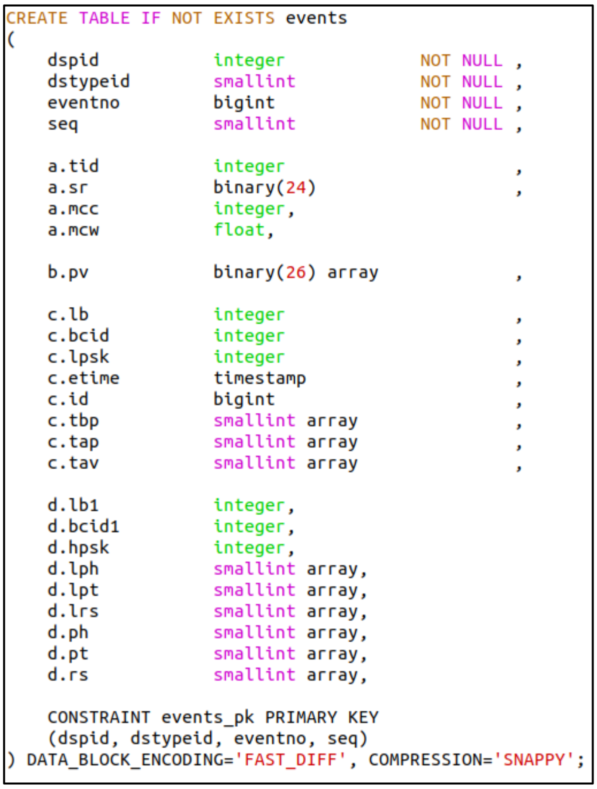
Families represent related data that is stored together on the file system. Therefore, all column family members should have the same general access pattern. The current defined families are:

- A: Event location (and MC info),
- B: Event provenance (from the processing history),
- C: Level 1 trigger (L1),
- D: High Level Trigger (L2 and EF for Run 1 data and HLT from Run 2 onwards).

In the *Event Location* family, we store information of the location of the event. It includes the *tid* (production task identifier) of the original dataset, allowing easy dataset deletion and ingestion crosschecks. It also includes the self-reference of the physical location of the event, as a 24-byte binary value, with 16 bytes representing the GUID identifier of the file that contains the event, and the OID1 and OID2 fields with 4 bytes each.

In the *Event Provenance* family, we store the chain of processing steps for this event. It is very similar to the previous self-reference field, but adding stream information using dataTypeFormat and dataTypeGroup. It is an array of records, where each record is represented as a 26-byte binary value (2 bytes for the dataType, and 24 bytes for the reference).

The other families maintain the trigger information, which comprises the bulk of the payload, and are divided into Level 1 and High-Level Trigger (EF or HLT) plus Level 2 for Run 1 data, including *lb* (luminosity block) and *bcid* (bunch crossing identifier).



```
CREATE TABLE IF NOT EXISTS events
(
    dspid           integer             NOT NULL ,
    dstypeid        smallint            NOT NULL ,
    eventno         bigint              NOT NULL ,
    seq             smallint            NOT NULL ,

    a.tid           integer                      ,
    a.sr            binary(24)                   ,
    a.mcc           integer,
    a.mcw           float,

    b.pv            binary(26) array             ,

    c.lb            integer                      ,
    c.bcid          integer                      ,
    c.lpsk          integer                      ,
    c.etime         timestamp                    ,
    c.id            bigint                       ,
    c.tbp           smallint array               ,
    c.tap           smallint array               ,
    c.tav           smallint array               ,

    d.lb1           integer,
    d.bcid1         integer,
    d.hpsk          integer,
    d.lph           smallint array,
    d.lpt           smallint array,
    d.lrs           smallint array,
    d.ph            smallint array,
    d.pt            smallint array,
    d.rs            smallint array,

    CONSTRAINT events_pk PRIMARY KEY
    (dspid, dstypeid, eventno, seq)
) DATA_BLOCK_ENCODING='FAST_DIFF', COMPRESSION='SNAPPY';
```

Figure 1: Schema of the Events table in Phoenix/Hbase.

3.2 Auxiliary tables

We need other tables to keep the dataset generated identifiers and bookkeeping data, as well as other related information. Currently we use these tables:

- *Datasets table*: stores information for quick dataset location, and related generated identifiers like the *dspid*. It also contains bookkeeping information relative to the status of the dataset during importing or ingestion phases, including the start and ending times. In addition it contains metadata information like the number of events, including computed information like the number of unique or duplicated events.

- *Data types table*: stores information and numerical identifiers about the data type formats and data type groups for derivations, as they will be used in the *dstype* field in the row key, and other fields in the data location and provenance families.

3.3 Data import procedure

The current production data has to be imported to Hbase/Phoenix with the new defined schema, so tools have been developed to perform this task when the new system enters production. The current tools can use Map/Reduce or Spark jobs to do the data loading, conversion, and writing the data in the Events table. They also write bookkeeping information of the import process in the Datasets table. Loading and check-pointing can be done at the individual file, dataset or container level, or even complete project or campaign.

As *tid* values for the datasets are needed, and they are not available in the currently stored files, we defined a new auxiliary table (*Dsguids*) to retrieve a *tid* by looking for a GUID. This table contains data exported from the Supervisor database for the production data and will be only needed during the data import procedure.

4. Trigger counter

The Trigger Counter is a web service able to provide information about the trigger composition of a given dataset. It is able to perform three different operations:

- It can count the occurrences of each trigger in the dataset. The counts can be grouped by Luminosity Block (LB) or Bunch Crossing IDentifier (BCID) if desired.
- It can provide a list of events, given trigger-based search criteria.
- It can calculate the overlap of the triggers in the dataset.

When asking for any of these operations, it is possible to apply a filter: a logical expression that will be applied to each event. The logical expression can include parentheses, boolean operators, trigger names or numbers and comparison operators to restrict integer ranges for LB and BCID.

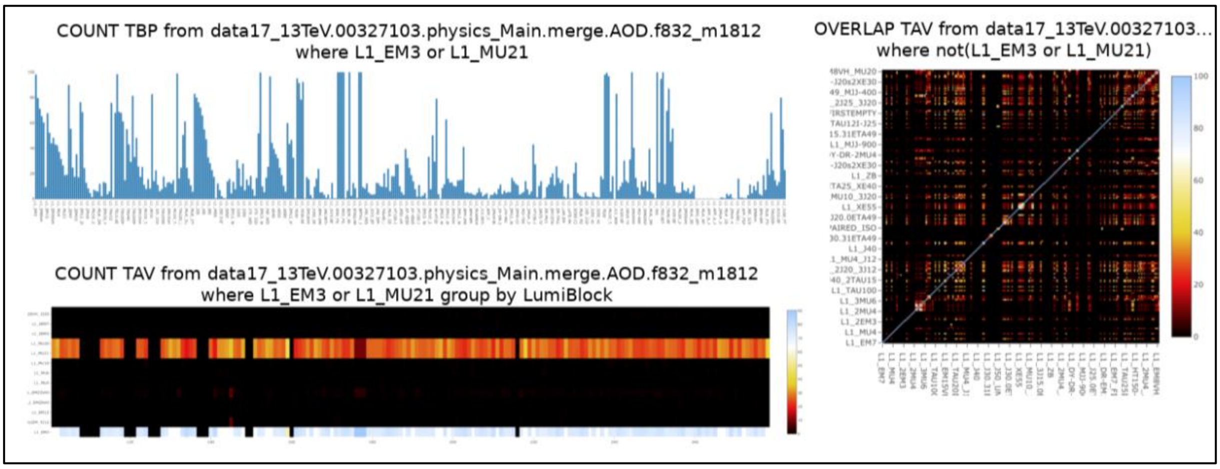


Figure 2 Left: Examples of displays of global trigger counts and counts grouped by luminosity block number. Right: The heat map of trigger overlaps within a given dataset. "TBP" and "TAV" refer to different stages of trigger processing: before prescale factors are applied, and after all vetoes are applied.

Since Trigger Counter is a user interactive web service, it must be reasonably fast when responding to the required operation. For this reason the data processed by Trigger Counter have been thoughtfully and thoroughly optimized to provide high performance when processing a dataset while keeping it as small as possible packing Level 1 triggers (512 possible values) into 10 bits using 32-bit words and HLT triggers (4 096 possible values) into 12 bits using 64-bit words and using LZ4 [20] to further compress the output which offers a satisfactory compression ratio while its decompression is fast.

Once the user has filled the form a Map/Reduce job is run and within a couple of minutes, the results are displayed in interactive plots. Additionally, JSON and CSV versions of the results are also available for download. Figure 2 shows some screenshots of the Trigger Counter data displays.

Trigger Counter is going to be migrated to make use of the Hbase/Phoenix infrastructure. The new data structures have fields and families to store the six trigger masks of the event. Some of the optimizations that were gained with the current implementation will be lost in favour of standard data structures offered by Phoenix.

5. Performance

We developed a new back-end plugin for HBase/Phoenix ingestion that can be used with the Consumer part of the Data Collection architecture. Input data is stored in the Object Store and read by Consumers that make the necessary schema adaptations to the data, using then the JDBC[1] Phoenix driver to store the data in HBase with the current schema. The first simple test inserting 48 datasets coming from the 2018 Tier-0 production showed a baseline performance of 3 kHz (3 000 events/s) per single-thread consumer. This implementation is still far from the 15 kHz of the previous Hadoop consumer plugin, but it must be taken into account that the first test did not use multiple threads, and that the event table design was slightly different without optimizations.

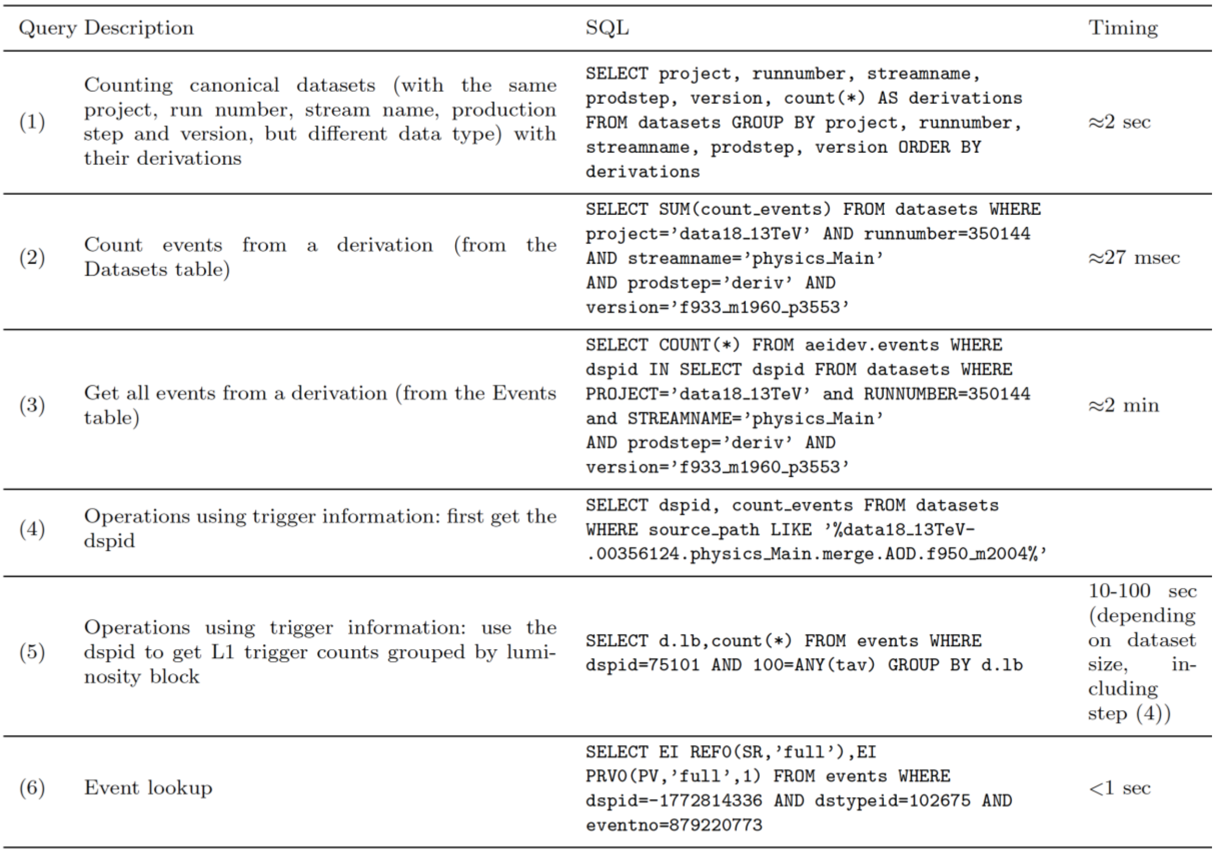
The current approach to the import of existing production data is to use Map/Reduce or Spark jobs to import all data from the Hadoop MapFiles. We can submit importer jobs with the desired input paths: from a single dataset to all the project data. The job analyzes the input data and spawns a single task per dataset (each dataset is stored in a single MapFile).

The first tests running Map/Reduce jobs used the CERN Analytix production cluster with 32 HBase region servers, the proposed event schema table with the described rowkey and four families of data, and Phoenix features like automatically salted keys (10 buckets), FastDiff encoding and Snappy compression (low performance overhead and provides space savings). We first tested individual dataset ingestion performance with a simple job using one Yarn container, one VCPU, and one mapper (no file splits). We ingested datasets in the order of 1 MB, 100 MB, 1 GB, and 10 GB, and yielded ingestion rates varying from 500 Hz (1 MB dataset) to 4.2 kHz (1 GB datasets). The performance of massive ingestion was measured by ingesting 8 000 datasets, containing 70 billion events, with a mean rate of 115 kHz. The procedure lasted one week in the shared cluster, and at some point 1 000 concurrent containers were running with 4 TB of allocated memory (which corresponds to 20% of the Analytix cluster). The big majority of datasets were imported correctly. Some tasks were killed by Yarn pre-empting the container to submit higher priority tasks. In this case the procedure automatically restarted and produced the correct output and book-keeping records. There was an issue also with large datasets that were taking more than 24 hours. Tasks can run more than that, but when finishing the task, the JDBC driver closes the connection prematurely, losing the last batch of data (order of 100 events). This is a problem of the Phoenix driver that we still need to solve.

Queries to check the use cases were performed on different instances, first on a standalone test cluster, and later on the Analytix production cluster:

- *Datasets table*: We can use the datasets auxiliary tables to discover datasets using partial dataset name components. We can also operate on the metadata stored per dataset, like for example the number of events. Even a full table scan will perform fast, so for example obtaining the total number of stored events summing up all the dataset entries in the entire ATLAS database will take less than one second.
- *Events table*: Then we can perform queries on the events table, and for these we run the set of queries in two scenarios:
    - 1st batch of queries while inserting data (100k ops/s writing load)
    - 2nd batch of queries without writing load (on the day after the 1st batch)

[1] Java DataBase Connectivity, the Java API to the database.



Query Description		SQL	Timing
(1)	Counting canonical datasets (with the same project, run number, stream name, production step and version, but different data type) with their derivations	`SELECT project, runnumber, streamname, prodstep, version, count(*) AS derivations FROM datasets GROUP BY project, runnumber, streamname, prodstep, version ORDER BY derivations`	≈2 sec
(2)	Count events from a derivation (from the Datasets table)	`SELECT SUM(count_events) FROM datasets WHERE project='data18_13TeV' AND runnumber=350144 AND streamname='physics_Main' AND prodstep='deriv' AND version='f933_m1960_p3553'`	≈27 msec
(3)	Get all events from a derivation (from the Events table)	`SELECT COUNT(*) FROM aeidev.events WHERE dspid IN SELECT dspid FROM datasets WHERE PROJECT='data18_13TeV' and RUNNUMBER=350144 and STREAMNAME='physics_Main' AND prodstep='deriv' AND version='f933_m1960_p3553'`	≈2 min
(4)	Operations using trigger information: first get the dspid	`SELECT dspid, count_events FROM datasets WHERE source_path LIKE '%data18_13TeV-.00356124.physics_Main.merge.AOD.f950_m2004%'`	
(5)	Operations using trigger information: use the dspid to get L1 trigger counts grouped by luminosity block	`SELECT d.lb,count(*) FROM events WHERE dspid=75101 AND 100=ANY(tav) GROUP BY d.lb`	10-100 sec (depending on dataset size, including step (4))
(6)	Event lookup	`SELECT EI REF0(SR,'full'),EI PRV0(PV,'full',1) FROM events WHERE dspid=-1772814336 AND dstypeid=102675 AND eventno=879220773`	<1 sec

Table 1: Examples of queries to the HBase/Phoenix tables and indicative performance.

Table 1 shows a few examples of typical queries to the Datasets and Events tables and their current performance. *Count* operations can be executed directly on the Datasets table and yield very fast results, as in examples (1) and (2).

A scan on the Events table is much slower, as query (3) on this table lasts over two minutes (151 s in the first batch and 140 s in the second batch). Note that this SQL query needs to use the dspid retrieved from the Datasets table, but this is a fast operation, and the bulk of the time is spent scanning the table with the dspid prefix.

For operations using trigger information, first we get the dspid from a particular entry of the Datasets table with query (4), then we use the previously retrieved dspid to select all the events from this dataset that also have (for example) the trigger 100 on the TAV (trigger after veto) mask, as in query (5). On the pre-production system this query lasted 107 seconds on the first batch, and 96 seconds on the second batch, but on the test machine with less data and no other users this test lasted only 12 seconds.

Finding a particular event requires knowledge of the dataset (dspid), data type and event number. On the test instance, this was done retrieving the reference and provenance, and decoding it with user-defined functions (EI_REF0, EI_PRV0) installed in the cluster; altogether query (6) lasted less than one second. When this test was performed, this query was not possible on the production infrastructure as it was not possible to install dedicated functions for users.

Although the queries done while writing data take more time to complete, it is not a substantial amount of time. In addition, this is the expected scenario on the future production system, where we will be ingesting data constantly.

4. Conclusions

The ATLAS EventIndex was designed to hold the catalogue of all ATLAS events in advance of LHC Run 2 in 2012-2013, and all system components were developed and deployed in their first implementation by the start of Run 2 in 2015. Like any software project, it went through several stages of development and optimisation through the years. Thanks to the partitioned project architecture, each new component version could be tested in parallel with the production version and phased in when its performance was considered stable, and better than the previous version. The EventIndex operation and performance during and after the Run 2 period have been satisfactory.

The significant increases in the data rates expected in LHC Run 3 and the subsequent HL-LHC runs require a transition now to a new technology for the main EventIndex data store. A new prototype based on HBase event tables and queries through Apache Phoenix has been tested and shows encouraging results. A good table schema was designed, and the basic functionality is ready. We are now working towards improved performance and better interfaces, with the aim to have the refactored system in operation well in advance of the start of Run 3 in 2022. According to our expectations, this system will be able to withstand the data production rates foreseen for LHC Run 4 and beyond.

References

[1] ATLAS Collaboration 2008 The ATLAS Experiment at the CERN Large Hadron Collider, JINST 3 S08003 doi:10.1088/1748-0221/3/08/S08003

[2] 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

[3] Hadoop and associated tools: http://hadoop.apache.org

[4] Hadoop MapFile: https://hadoop.apache.org/docs/r2.6.2/api/org/apache/hadoop/io/MapFile.html

[5] Baranowski Z et al. 2017 A study of data representation in Hadoop to optimise data storage and search performance for the ATLAS EventIndex, J. Phys.: Conf. Ser. 898 062020, doi:10.1088/1742-6596/898/6/062020

[6] Kudu: http://kudu.apache.org

[7] Baranowski Z et al. 2019 A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage, EPJ Web of Conferences 214, 04057, doi:10.1051/epjconf/201921404057

[8] HBase: https://hbase.apache.org/

[9] Phoenix: https://phoenix.apache.org

[10] Fernandez Casani A et al. 2019 Distributed Data Collection for the Next Generation ATLAS EventIndex Project, EPJ Web Conf. 214 04010, doi:10.1051/epjconf/201921404010

[11] Zaharia M et al. 2016 Apache Spark: a unified engine for big data processing, Commun. ACM 59, 11, 56–65. doi:10.1145/2934664

[12] A Python client library for accessing messaging servers using the STOMP protocol: https://github.com/jasonrbriggs/stomp.py

[13] An Amazon Web Services (AWS) Software Development Kit (SDK) for Python: https://github.com/boto/boto3

[14] Google Protocol Buffers (Google's Data Interchange Format): http://code.google.com/apis/protocolbuffers

[15] Mesnier M, Ganger G R and Riedel E 2003 IEEE Communications Magazine 41 84–90, ISSN 0163-6804

[16] EOS: https://eos-docs.web.cern.ch

[17] Barreiro Megino F H et al. 2017 PanDA for ATLAS distributed computing in the next decade, J. Phys. Conf. Ser. 898 052002. doi:10.1088/1742-6596/898/5/052002

[18] Crc16: https://encyclopedia2.thefreedictionary.com/Crc16

[19] GUID: Global Unique IDentifier, https://en.wikipedia.org/wiki/Universally_unique_identifier

[20] LZ4: https://en.wikipedia.org/wiki/LZ4_(compression_algorithm)