Hindawi
Computational and Mathematical Methods
Volume 2023, Article ID 6900908, 19 pages
https://doi.org/10.1155/2023/6900908

Research Article

Big Data Analytics for the ATLAS EventIndex Project with Apache Spark

Álvaro Fernández Casaní, Carlos García Montoro, Santiago González de la Hoz, José Salt, Javier Sánchez, and Miguel Villaplana Pérez

Institute of Corpuscular Physics-IFIC (CSIC/UV), E-46980 Paterna, Spain

Correspondence should be addressed to Álvaro Fernández Casaní; alvaro.fernandez@ific.uv.es

Received 25 September 2022; Revised 8 September 2023; Accepted 14 September 2023; Published 27 September 2023

Academic Editor: Higinio Ramos

The ATLAS EventIndex was designed to provide a global event catalogue and limited event-level metadata for ATLAS experiment of the Large Hadron Collider (LHC) and their analysis groups and users during Run 2 (2015-2018) and has been running in production since. The LHC Run 3, started in 2022, has seen 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. A new core storage service is being developed in HBase/Phoenix, and there is work in progress to provide at least the same functionality as the current one for increased data ingestion and search rates and with increasing volumes of stored data. In addition, new tools are being developed for solving the needed access cases within the new storage. This paper describes a new tool using Spark and implemented in Scala for accessing the big data quantities of the EventIndex project stored in HBase/Phoenix. With this tool, we can offer data discovery capabilities at different granularities, providing Spark Dataframes that can be used or refined within the same framework. Data analytic cases of the EventIndex project are implemented, like the search for duplicates of events from the same or different datasets. An algorithm and implementation for the calculation of overlap matrices of events across different datasets are presented. Our approach can be used by other higher-level tools and users, to ease access to the data in a performant and standard way using Spark abstractions. The provided tools decouple data access from the actual data schema, which makes it convenient to hide complexity and possible changes on the backed storage.

1. Introduction

The Large Hadron Collider (LHC) [1] is the biggest particle accelerator built and is located at CERN, the European Organization for Nuclear Research, on the border of Switzerland and France. It is found in a circular tunnel with a length of 27 kilometres and 100 metres underground.

ATLAS [2] is one of the four big detectors registering particle collisions, or events, and is devoted to testing the predictions of the standard model of particle physics and to physics beyond the standard model and the development of new theories to describe our universe. Collisions are produced at a rate of 40 MHz, or with a bunch spacing of 25 ns, which would mean storing 60 TB/s. A multilevel online trigger system selects the most interesting events to effectively reduce the data recording rate up to a manageable order of 1 kHz and the data rate of gigabytes. Data is recorded at CERN Tier-0, and then distributed with grid technologies to 70 computing sites worldwide. These are organized in 10 Tier-1 centres to store the RAW data, with smaller Tier-2 and Tier-3 centres for reprocessing and analysis tasks. In addition to real data, around three times of simulated events with Monte Carlo production method are generated and stored.

A catalogue of all data, real and simulated, is needed to search for events by several criteria and to provide production checkings and analytics insights. ATLAS EventIndex [3, 4] was developed to provide this metadata catalogue and was put in production during Run 2 (2015-2018), indexing more than 30 billion (10^9) events, with 1/4 real and 3/4 simulated data. In the following runs of the experiment with run 3 (2022-2025) and run 4 (high-luminosity LHC, 2029 onwards), it is expected an increase up to a factor 10 in the production rate, reaching 100 billion real events and 300 billion events produced.

A new implementation of the system capable of absorbing increasing data rates and able to fulfil the required use cases is being implemented using the HBase [5] as the main and unique storage. An Apache Phoenix [6] layer provides SQL-like access for transactional and extraction queries. New tools are being developed providing analytics over large quantities of data.

This paper presents our contributions in the data access area, adding interactive data access and analysis, which was not possible with the previous model. We have improved analytical use cases with the EventIndex data stored in HBase/Phoenix, and accessing with a platform based on Spark [7], using its abstractions and a set of analytical tools implemented in Scala. The new tool and algorithms presented solve data access for our application use cases in areas like data discovery, duplicate detection, and overlap calculation among datasets that are now integrated. Data discovery capabilities produce Spark DataFrames usable by the rest of the tools. In addition, data and results might be maintained in cache, which was not possible before, allowing algorithm chaining and improving overall resource usage. The new contributed overlap calculation algorithm has computational cost O(n) with the number of events and spatial cost O(s^2) with the number of streams.

With our tools, we abstract the backend data model, decoupling the data access from the actual data schema and the selected technologies. This approach is very convenient to hide model complexity with an accessible defined interface for tools and users. It also masks possible changes in the data model, which are invisible for the user as the defined interfaces do not change.

The rest of the article is organized as follows. Section 2 shows the requirements and current use cases. Section 3 details the architecture of the proposed EventIndex analytics platform. Section 4 discusses the implementation of the data access layer and algorithms to solve the data discovery, duplication detection, and event overlap calculation matrices using Spark Abstractions. Finally, Section 5 ends with the conclusions about the presented work.

2. Requirements and Use Cases

The main use case when the EventIndex project was started was the selection of particular events, or event picking over a large catalogue of event metadata.

Later, one more analytical case of the EventIndex project was included, with the detection of particular patterns over large quantities of data. Data discovery mechanisms are required to provide access to selected data based on user-defined constraints.

A group of use cases are related to data consistency checks. ATLAS production processes can temporarily fail, producing duplicate events with the same identifiers. Simulated Monte Carlo [8] procedures are also vulnerable to generating incorrect data, so detection methods are needed. Detection of these duplicate event data is required at different granularities, including complete datasets or containers.

The ATLAS derivation framework [9] outputs the selected events that are requested by physics analysis groups. It is useful to detect the event overlaps among the derived datasets, identifying them to optimise the procedures and used resources.

Studies over the stored metadata information are anticipated. Trigger overlap studies within a dataset or among derived datasets are valuable. The objective is to identify the trigger chain pairs that were fired simultaneously at event recording, which can provide useful statistics and insights about the trigger processes.

Other use cases requiring analysis over large quantities of data can arise in the future, so general access methods are supported.

A new data access layer is required to leverage the back-end data storage improvements being developed [10]. To support data retrieval, a new SQL-like interface opens the possibility of integration with JDBC protocol. Previous web front-ends designed to access relational data back-ends can be rapidly adopted. In addition, a new low-latency access framework is needed to support the analytic use cases with semantics expressed in higher-level languages, instead of the restricted SQL syntax.

3. EventIndex Analytics Platform Spark

The EventIndex analytics platform provides services for solving OLAP (online analytical processing) use cases and obtaining insights about the data. Figure 1 shows the proposed architecture which is based on Apache Spark [7], an engine for large-scale data analytics that provides abstractions for data modelling and in-memory efficient operations. It can be accessed interactively with command-line consoles or web interfaces like notebooks. A programmatic interface is also available, making it easy to run background processes on the provided resources.

Spark interfaces natively with resource management tools like in our case YARN [11] to provide access to the CERN cluster, which comprises dozens of machines that host data and computing servers.

Data storage for EventIndex is implemented in HBase [5] to provide a unique and unified backend for all data and use cases. HBase is a large-scale distributed key-value storage included in the class of NoSQL big data stores. It suits our application use case as it can scale and store petabytes of data with our ingestion rate.

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

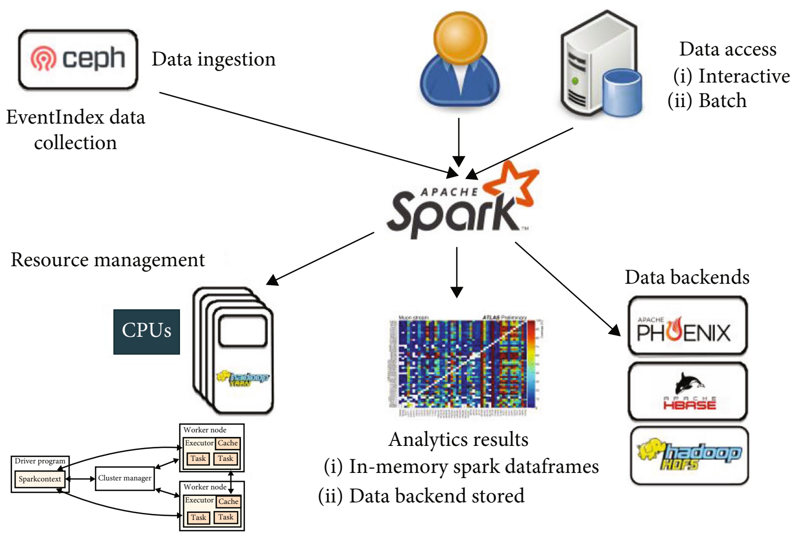


Figure 1: Architecture of the EventIndex analytics platform based on Apache Spark.

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 separate use cases. Further analytic use cases with larger amounts of data are not foreseen but can still be achieved by running MapReduce/Spark jobs on the Hbase files, as they are stored in the Hadoop Filesystem HDFS.

Apache Phoenix [6] is a layer over HBase that enables SQL access and provides an easy entry point for users and other applications. Although HBase is schema-less storage, Phoenix requires a schema and data typing to provide its SQL functionality. Schema versioning and dynamic late binding for the same tables are also supported. In EventIndex, our data rarely varies its schema, so we can benefit from Phoenix designing the required schema and tables accordingly.

We use Apache Spark as a framework for dealing with data in a distributed manner that provides some abstractions that are very useful and performant.

The most important is the Spark DataFrame, which represents a set of data that might be residing on several physical nodes and that allows to apply and chain operations to produce other DataFrames or store them in multiple backend storage kinds.

One important characteristic is that operations are applied lazyly in memory, only when it is needed, so optimization can be done dynamically on chained operations. It is failure resilient, so the computations can be reapplied automatically in case a node fails.

Scala is the language that Spark was written in and is the most supported programming language to access all Spark APIs. It is running in the JVM (Java virtual machine), so Java classes can be called from Scala, with the benefits of a concise and high-level language.

Spark and Scala are used in this work to access the backend HBase/Phoenix storage data with the defined EventIndex data model.

In the following sections, we discuss the data model implemented on HBase/Phoenix and the data ingestion procedures that were done to support the development and evaluation of the data access analytical tools.

3.1. Data Model.

3.1. Data Model. The data structures in the EventIndex HBase storage use a defined Phoenix schema [12].

The event data resides in a big table with billions of entries that represent all physics events produced by the ATLAS experiment. In the ATLAS data and distributed production model, events are stored in files (GUIDs), grouped in datasets (TID), and datasets are grouped in containers. Therefore, these relations are expressed using additional tables.

A logical entity-relationship model of the EventIndex data is represented in Figure 2, with entities explained in detail in the following subsections.

The original ATLAS dataset nomenclature [13] includes several fields to identify them, which in general are *Project*, *runNumber*, *streamName*, *prodStep*, *dataType*, and *version*. Due to the grid-distributed production system, events belonging to a physics container are divided into datasets that can be referenced within the same container info and an additional identifier (*TID*). The events table is linked to the dataset and container tables by means of a constructed composite foreign key that includes the dataset and data type identifier.

3.2. Event Table. In our schema, we have a big table that stores all event records. A small quantity of metadata is stored per event, with information including event identification (for real and simulated events and some LHC conditions during event recording), location information (where to find the event in the distributed files within the grid), and trigger information (about the conditions that flagged and selected a particular event detected by ATLAS to be permanently stored) [14].

A representation of the events table can be found in Figure 3. In this schema, every entry is defined by a row key and by values grouped in several families.

Row key: in HBase, the best performance is obtained when searching for data using row keys. Random access to

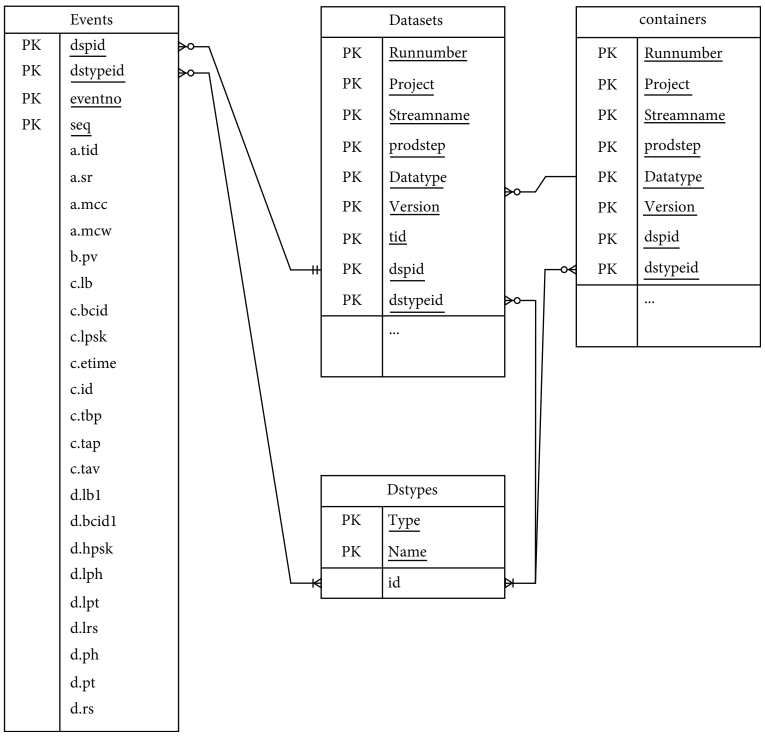


Figure 2: Logical entity-relationship model of the EventIndex data.

the complete key is translated into an HBase to get excellent results. A range scan over a prefix of the key is also a common operation. The EventIndex schema is designed to include the most-accessed information to solve the required use cases in the key, leaving extra information in the value. Searching by value is internally transformed in the full HBase scan, which has lower performance. Having billions of entries means that keeping the key length to a minimum is important both for performance and for total occupied volume reasons.

The row key is a 16-byte binary composed internally of several parts that can be used for prefix search: dspid.dstype.eventno.seq.

(i) dspid (integer: 4 bytes) is an identifier generated at ingestion time. Takes into account internally the dataset name, except the datatype. Therefore, datasets with the same (Project.runNumber.StreamName, prodStep.AMITag) will share the same dspid. This is intended to search by dspid as all different datatypes of a dataset will sit close in the backend storage, making the search performant for solving the datasets overlap the computation use case

(ii) dstype (smallint: 2 bytes) is the dataset identifier for the datype of the dataset name, not included in the dspid. This identifier is internally computed using dataTypeFormat (5 bits = 32 values) and dataTypeGroup (11 bits = 2048 values) as defined in the dataset nomenclature [13], for optimal usage

(iii) eventno (long: 8 bytes) is the event number

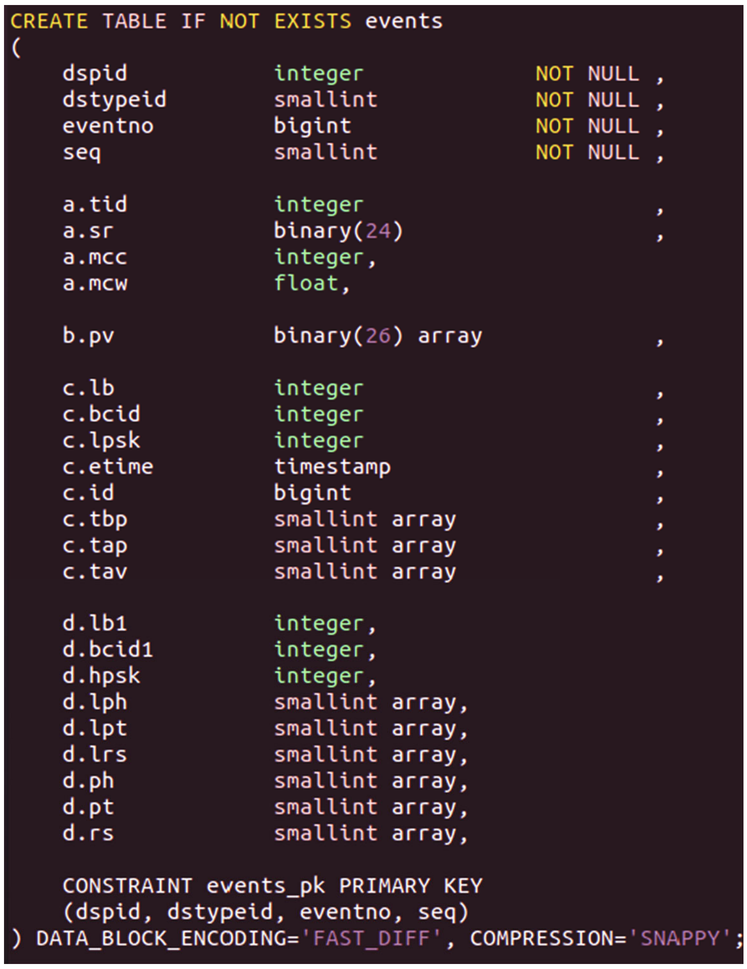
(iv) seq (short: 2 bytes) is the sequence used to deduplicate event entries when the EventNumber collides. It makes the row key unique in case of datasetName and EventNumber duplication and is computed as the crc16 value of (GUID:OID1-OID2) which is unique. The GUID [15] is an identifier of the file containing the event, and OID1-OID2 are the internal pointers within that file. The probability of key clashing within the same dataset, involving two or more equal computed hashes out of a group of n (being n = 2^16 with crc16 algorithm), can be calculated with what is known as the birthday problem [16]. This probability is estimated to be low enough in our production system to not cause problems

The values are grouped in 4 families that contain related data and have the same general access pattern. In this way only required data is accessed for solving the required use cases. Family A provides the event location information (and Monte Carlo information for simulated data) to solve event lookup and pick use cases. family B contains the event provenance and provides information for data lineage used in data quality checking. The last 2 families provide trigger information that is usually accessed separately: family C for the level 1 trigger, and family D for the rest of the trigger levels.

The fields for each family can be found next:

(i) *Family A*. Event location information (and Monte Carlo information for simulated data).

(ii) *TID (integer: 4 bytes) production task identifier*. The numeric value found in the dataset name



```
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 3: Event table schema representation in Apache Phoenix grammar [12].

suffix "_tidNNNNNNNN_X" for this kind of TID dataset, or 0. This further identifies the part of the dataset that it belongs to, for data bookkeeping purposes

(a) *sr (binary: 24 bytes) self-reference*. A binary composed of the GUID (16 bytes) file identifier where the event is found, OID1 (4 bytes), and OID2 (4 bytes) representing pointers inside the file to locate the actual event

(b) *mcc (integer: 4 bytes) Monte Carlo channel number*. In case of simulated events, not used otherwise

(c) *mcw (float: 4 bytes) Monte Carlo weight*. In case of simulated events, not used otherwise

(iii) *Family B*. Event provenance provides information about data lineage

(a) *pv (binary array: 26 bytes per entry) provenance*. A binary array with one entry per element of the data lineage in the production chain of the event as found in the analysed file. Every entry is a 26-byte binary composed by the dstype (2 bytes) and the self-reference (24 bytes) as previously presented

(iv) *Family C*. Level 1 (L1) trigger information [14]

(a) *lb (Integer: 4 bytes)*. Luminosity block

(b) *bcid (Integer: 4 bytes)*. Bunch crossing identifier

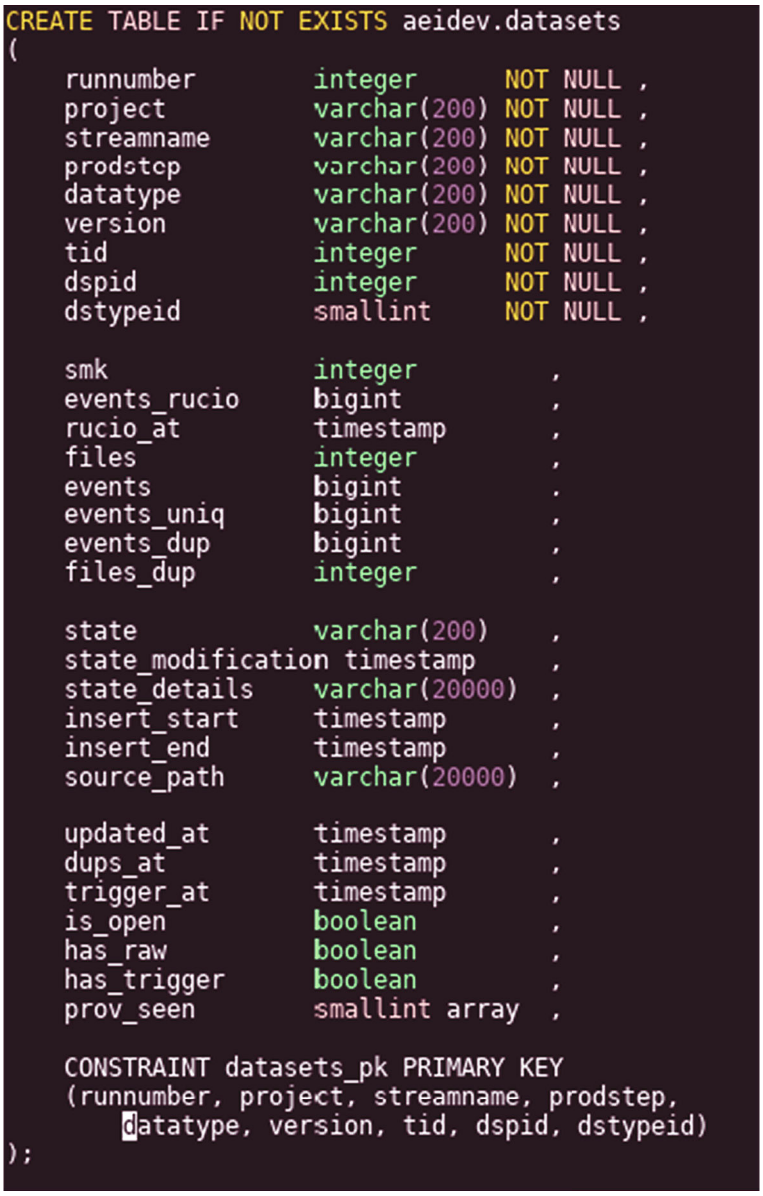
(c) *lpsk (Integer: 4 bytes)*. L1 trigger prescaler key

(d) *etime (timestamp: 16 bytes)*. Using the Java Timestamp type with an internal representation of the number of nanoseconds from the epoch

(e) *id (bigint: 8 bytes)*. L1 trigger id

(f) *tpb (smallint array) trigger before prescaler*. A variable-length array of smallint (2 bytes) entries

(g) *tap (smallint array) trigger after prescaler*. A variable-length array of smallint (2 bytes) entries



```
CREATE TABLE IF NOT EXISTS aeidev.datasets
(
    runnumber         integer       NOT NULL ,
    project           varchar(200)  NOT NULL ,
    streamname        varchar(200)  NOT NULL ,
    prodstep          varchar(200)  NOT NULL ,
    datatype          varchar(200)  NOT NULL ,
    version           varchar(200)  NOT NULL ,
    tid               integer       NOT NULL ,
    dspid             integer       NOT NULL ,
    dstypeid          smallint      NOT NULL ,

    smk               integer          ,
    events_rucio      bigint           ,
    rucio_at          timestamp        ,
    files             integer          ,
    events            bigint           .
    events_uniq       bigint           ,
    events_dup        bigint           ,
    files_dup         integer          ,

    state             varchar(200)     ,
    state_modification timestamp       ,
    state_details     varchar(20000)   ,
    insert_start      timestamp        ,
    insert_end        timestamp        ,
    source_path       varchar(20000)   ,

    updated_at        timestamp        ,
    dups_at           timestamp        ,
    trigger_at        timestamp        ,
    is_open           boolean          ,
    has_raw           boolean          ,
    has_trigger       boolean          ,
    prov_seen         smallint array   ,

    CONSTRAINT datasets_pk PRIMARY KEY
    (runnumber, project, streamname, prodstep,
        datatype, version, tid, dspid, dstypeid)
);
```

Figure 4: Datasets table schema representation in Apache Phoenix grammar. Represents a dataset and summarises information of the event table.

(h) *tav (smallint array) trigger after veto*. Avariable-length array of smallint (2 bytes) entries

(v) *Family D*. Level 2 (L1) and event filter (EF) trigger information for run 1/high-level trigger (HLT) information for run 2 onwards. [14]

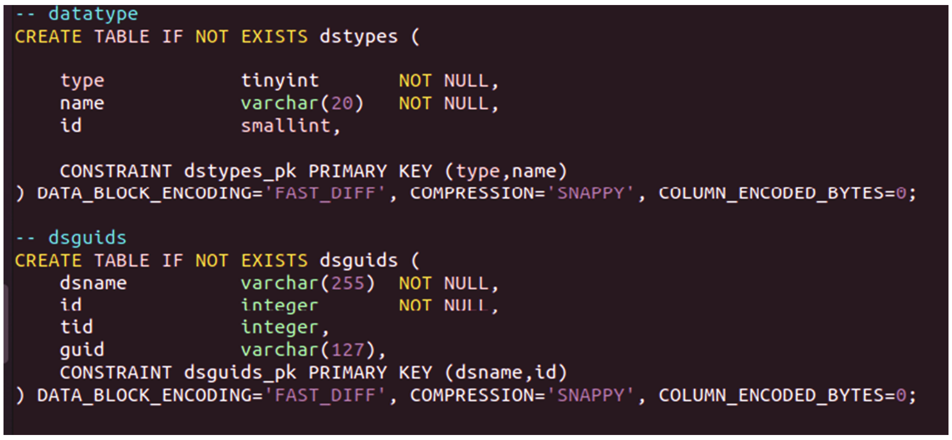
(a) *lb1, bcid1 (integer: 4 bytes)*. contains the same values as the counterpart in family C, named with a suffix due to a limitation in Apache Phoenix Spark to access the same field names in different families

(b) *hpsk (integer: 4 bytes)*. HLT triggers prescaler key

(c) *lpH, lpt, lrs (smallint array)*. Level 2 (L2) physics, passthrough, and resurrected variable length of arrays

(d) *ph, pt, rs (smallint array)*. HLT physics, passthrough, and resurrected variable length of arrays

3.3. Dataset Table. We are defining other tables that contain metainformation about the events stored and grouped in datasets, that are needed for data discovery. The dataset table can be found in Figure 4. It caches available information at the time of ingestion, including summary or bookkeeping data. In addition, it can store the calculated information during the analytical procedures as we will see later.



```
-- datatype
CREATE TABLE IF NOT EXISTS dstypes (

    type            tinyint       NOT NULL,
    name            varchar(20)   NOT NULL,
    id              smallint,

    CONSTRAINT dstypes_pk PRIMARY KEY (type,name)
) DATA_BLOCK_ENCODING='FAST_DIFF', COMPRESSION='SNAPPY', COLUMN_ENCODED_BYTES=0;

-- dsguids
CREATE TABLE IF NOT EXISTS dsguids (
    dsname          varchar(255)  NOT NULL,
    id              integer       NOT NULL,
    tid             integer,
    guid            varchar(127),
    CONSTRAINT dsguids_pk PRIMARY KEY (dsname,id)
) DATA_BLOCK_ENCODING='FAST_DIFF', COMPRESSION='SNAPPY', COLUMN_ENCODED_BYTES=0;
```

Figure 5: Auxiliary tables representation for the dataset types (dstypes) and file GUIDs (DSGUID).

Row key is composed by all the string fields that compose the dataset name [13], namely, project, streamname, prodstep, datatype, and version. It also includes the production task identifier (TID) for those datasets that include them, 0 in other cases.

Dspid and dstypeid refer to the values stored in the event table and is the way to link these two tables. As a useful side effect, it defines a canonical dataset container, which includes all events that share the same dspid.dstypeid [10], grouping all dataset fields but TID.

The rest of the field are values of the entry that contain metadata about the dataset:

(i) *Caching or computed values. smk, events, events_uniq, events_dup, files_dup, prov_seen*

(ii) *Booleans identifying characteristics of the dataset. is_open, has_raw, has_trigger*

(iii) *Information and timestamps about the bookkeeping of this dataset. state, state_details, state_modification, source_path, insert_start, insert_end, updated_at, dups_at, trigger_at*

We will see the usage of many of these fields in the following sections.

3.4. Container Table. We will maintain in this table a representation of a particular kind of container, grouping all information about entries in the dataset table that share the same dspid.dstypeid. Therefore, the structure of the table is basically the same as the datasets table, but the key does not include the TID. Therefore, an entry in this table represents a canonical dataset container [10], which might reflect one or more datasets in the datasets table.

The fields of an entry represent the sum of the fields of the related entries in the datasets table, or the calculated values with the container (canonical dataset) granularity.

3.5. Auxiliary Tables. Other auxiliary tables that help in the process of data identification, or during ingestion and importing, can be seen in Figure 5.

Data types table (dstypes): store names and numerical identifiers about the data type, distinguishing between data type formats (type = 0) and data type groups (type = 1). For every entry, we can find its "name" and its numerical "id." The id values will be used for building the dstype id field in the row key and other fields in the data location and provenance families.

Files table (DSGUIDs): this table relates information about the dataset name, the contained files (GUID), and their TID (production task identifier). It is needed because during the data ingestion procedure (see the following section), this data might not be available in the source, and we need to consult it in this table to fill in the proper events and datasets tables.

3.6. Evaluation Setup. To test the proposed data model and query tools, the first experiments were made to fill the data tables. The upper part of Figure 6 shows a new backend plugin (PhoenixWriter) that was developed for the consumer of the EventIndex collection system, to be able to fill the event tables in HBase/Phoenix directly with the distributed data collection [17].

In the right part of the figure, we are showing another method for massive ingestion of the previous production data stored in HDFS [18], using the proposed HBase/Phoenix data model. The ingestion procedure uses MapReduce [19] jobs to read the input production data, convert it to Phoenix Schema, and store it in HBase/Phoenix tables. These PhoenixImporters share code and use the same data mangling methods with the PhoenixWriter Consumer. The reason to use MapReduce jobs in the ingestion procedure instead of Spark was the good performance of HDFS bulk loading and that the development for this one-shot import procedure was faster and easily integrated with other established procedures. On the contrary, analytics on the data residing in Hbase/Phoenix will be done with Spark, as it provides more versatility as we will see in the next sections.

The input data resides in the HDFS file system within the production CERN Hadoop cluster. The output data will be in the proposed HBase/Phoenix schema, using the events and rest of the metatables which serve also for bookkeeping purposes of the ingestion procedure. The procedure itself

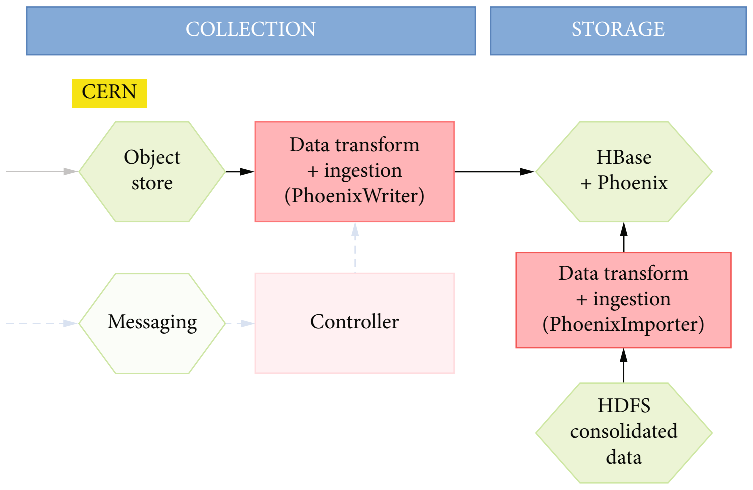


Figure 6: Architecture for the evaluation setup. Green hexagons represent data stores, and pink rectangles represent processes.

consists of the submission of MapReduce jobs to the Hadoop cluster, with the input data paths desired for granularity (individual dataset, entire project). The map-only tasks transform each event from the original format to the HBase/Phoenix schema, writing it in the events tables.

The additional data that is not available in the original format and is needed for the event keys (dspid, dstypeid) is read from the metatables. This is also what happens for the TID and GUID relations that might not be available in HDFS consolidated data.

This data will be provided by the Data Collection task in production, but for these tests, the data can be prefilled or generated dynamically by this ingestion procedure. The procedure looks for the needed entry and, if not available, uses a database sequence table to create it for subsequent data. The rest of the fields can be transposed directly to the new format or computed on the flight from the original input data.

We have implemented a checkpointing mechanism at the dataset level, being able to stop and restart the procedure. This procedure avoids restarting successful jobs unless an overwrite option is configured explicitly. We are writing bookkeeping data in the setup and cleanup MapReduce phases. These phases update the dataset tables with information about what data is imported (the logical dataset identifier, but also referring to the physical HDFS files), timestamps with starting and finishing times, status of the dataset, and dataset metadata (number of events, number of files, etcetera).

The test ingestion procedure fills the following fields:

(i) *State*. Inserting, done, fail, and writing state_details in case of failure

(ii) *Source path*. HDFS input path

(iii) *Timestamps*.state_modification, insert_start, and insert_end

(iv) *Number of events inserted*. Events

3.7. Massive Ingestion Test. A massive ingestion test campaign was done to check the response of the backend system and to have data for subsequent query tests. We used the CERN Analytix production cluster, which was composed of 39 nodes (32 Hbase region servers), with a total memory of 18 TB and 1658 vcores. It has to be noted that this is a shared cluster of multiple projects at CERN. Clusters included the following distributions: Hadoop 3.2.1 and HBase 2.2.4.

Ingestion experiment was run for 1 week, with input from several “project” areas in HDFS, with multiple MapReduce jobs sent:

(i) First batch of jobs to index real data from the year 2018 datasets (6,254 YARN tasks)

(ii) Second batch jobs to index real data from the year 2017 datasets (6,796 YARN tasks)

(iii) Other jobs to index some variety of datasets

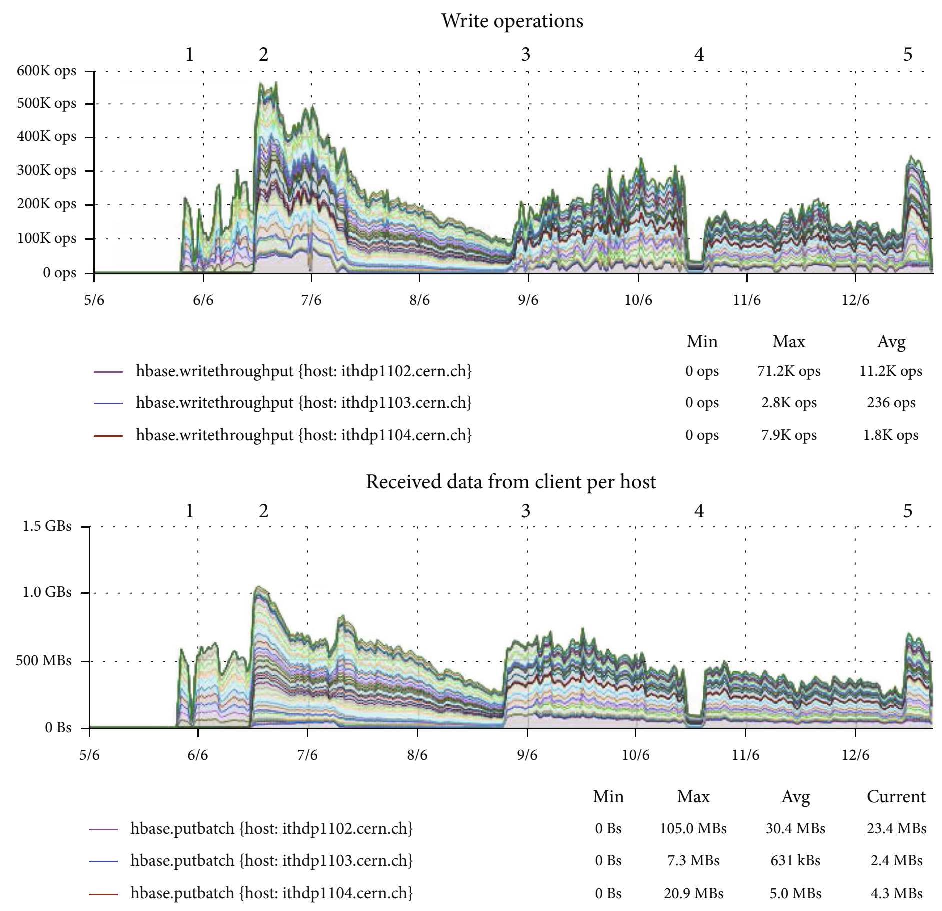
Event table was defined with the following options: DATA_BLOCK_ENCODING = ‘FAST_DIFF’, COMPRESSION = ‘SNAPPY’, SALT_BUCKETS = 10.

These parameters signal the table to use the default diff encoding and Snappy compression. In addition, we use 10 salt buckets to automatically include a prefix byte hashed, to distribute the load among regions and avoid hot spotting when using the same or monotonically increasing keys.

During the data ingestion experiment, we were monitoring several aspects of the ingestion, as we can see in Figure 7.

We can see that at point 1 (date 6/06), the write operations are evenly distributed to only 10 RS machines (the salted table key space is originally automatically presplitted in 10 region servers according to the configured salt bytes boundaries to ensure load distribution among region servers) reaching a maximum throughput of 300 Kwrites/s corresponding to 500 MB/s.

Later that day (point 2, midday 6/06), it was manually triggered to distribute some regions to the rest of the RS



	Min	Max	Avg
hbase.writethroughput {host: ithdp1102.cern.ch}	0 ops	71.2K ops	11.2K ops
hbase.writethroughput {host: ithdp1103.cern.ch}	0 ops	2.8K ops	236 ops
hbase.writethroughput {host: ithdp1104.cern.ch}	0 ops	7.9K ops	1.8K ops

	Min	Max	Avg	Current
hbase.putbatch {host: ithdp1102.cern.ch}	0 Bs	105.0 MBs	30.4 MBs	23.4 MBs
hbase.putbatch {host: ithdp1103.cern.ch}	0 Bs	7.3 MBs	631 kBs	2.4 MBs
hbase.putbatch {host: ithdp1104.cern.ch}	0 Bs	20.9 MBs	5.0 MBs	4.3 MBs

Figure 7: System load metrics over the duration of the experiment. Upper panel shows the write operations; the lower panel shows volume of received data.

machines, to further distribute the load among all region servers and reach up to 550 Kwrites/s and 1 GB/s.

The manual intervention was needed to avoid the slow start when there are no presplits. We have learned that we can avoid the slow start by applying several presplits, and then leaving HBase to continue automatically.

It seems that there is no direct answer for the optimal number of regions for a given load. The accepted recommendation is to start with a number of split multiples of the number of region servers. Then HBase automated splitting will do the next split by itself, with no slow start.

Point 3 (9/06) signals where the first batch of jobs to index data 18 datasets (6,254 tasks) are finishing, although the bigger datasets are still running. At this point, the batch of data 17 jobs are started.

At point 4 (midday 10/06), we killed all tasks and restarted them. It reduced the number of writes to about 200 Kwrites/s but maintained the performance of 500 MB/s.

At point 5 (afternoon 12/06), there was no change at the job side, but the queue configuration was changed by the CERN cluster managers. A dedicated YARN queue was created for EventIndex with complete access to a maximum of 1 k vcores and using up to 4 TB of memory. At this point, the performance was raised to 300 Kwrites/s and 750 MB/s.

Results show that the big majority of datasets were imported correctly, and in particular, there were no failures with corrupted data or bad records. There was also no problem in converting the previous data format to the new Phoenix schema. There were, however, some issues related to the job management inside the cluster. Some tasks are killed by YARN (container preempted by the scheduler), due to the priorities configured in the cluster. This is no problem for our procedure since, as we said, we have this into account in our bookkeeping, and already correctly finished tasks (all dataset events are written correctly in HBase) will not be written again. There is no data integrity problem writing the same data because restarting stops tasks, as the event keys are the same and the results will be idempotent. Yet this restarting means many CPU hours lost for rewriting the same data.

We also spotted an issue for tasks running over 24 hours. Tasks can run much longer without problem writing all data, but when closing the connection, the internal JDBC driver loses the last batch of data (order of 100 events). We tried to configure the relevant configuration option without success (set phoenix.client.connection.max.duration >24 H), and the investigations led to a problem in the JDBC usage in the Phoenix implementation, which was reported to the developers.

The final result and the measured resource usage on the cluster were the following:

(i) 2,347 HBase regions (continuous, sorted set of rows that are stored together), using the 10 buckets (prefix byte hashed in the key) originally configured

(ii) max resources: 1000 vcores, up to 977 concurrent YARN containers (where the application is run), and~4 TB of memory in total (20% of the cluster)

The output and results of the experiment on the events table:

(i) 7,941 datasets written

(ii) 70 billion events

(iii) Volume size of 22 TB. Distribution of the data in families:

  (A) Event location: 1.89 TB

  (B) Event provenance: 2.38 TB

  (C) L1 trigger: 9.25 TB

  (D) L2 and HLT physics trigger: 8.41 TB

(iv) 117 kHz mean insertion rate (periods with less activity due to some jobs finishing tasks, some hours off among insertions,…)

We can observe that the larger volume of data is occupied by the trigger information, followed by the provenance and event location information.

In the following sections, we will detail the implemented access data algorithms and the evaluation done using this ingested data.

4. Data Access and Analytic Algorithms

4.1. EventIndex Data Discovery. As we have seen in the data model, the EventIndex data resides in a big table linked to the dataset and container tables by means of a constructed composite key. To find the relevant event data for resolving defined use cases, or making any analytical study, we need a set of tools that can find the event data. This can be done by searching for any of the identification fields, or by any other fields that represent summary data for every dataset and container, including

(i) Number of total, unique, and duplicate events

(ii) Number of total files (GUIDs) and that contains duplicates (dataset granularity)

(iii) Data collection bookkeeping info: status of the dataset and date of updated information

(iv) Metadata about related info from the related events, basically if it contains raw data, trigger, and provenance details

We have produced relevant tools to look up data of interest that can be used later for further use cases. In particular, the higher entry functions are *findDatasets()* and *findCanonical()* that access the related tables and produce a Spark DataFrame with results that can be consulted and refined to all Spark operations. These data entities are defined on a *schema* with named columns that, in this case, reflect the underlying data schema of the backend HBase/Phoenix tables, making them available to the rest of operations.

The incarnation of the data is done with a *lazy evaluation* policy, so the results are only available when actions are called. Another advantage when modelling the data with Spark DataFrames is the possibility to apply *Spark SQL functions*, which allow the usage of SQL queries.

Some examples of the functionality are shown next. In the example in Figure 8, we show our tool working with the interactive spark shell, and we observe that we only need to import the "Util" class of the "eventindex.analitycs.spark" package to access the described functions. When calling, findCanonical() will produce the Spark DataFrame. The generic "count" function will give us the number of entries of the underlying data table, so in this example, there are 1132 container datasets. We can use the groupBy() function by the "project" named column, and then count and show how many datasets are available by project. Results are returned fast in less than a second, as it accesses only the summary dataset and container tables, orders of magnitude smaller compared with the event table.

In Figure 9, we intend to find datasets from a container dataset:

"mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.-deriv.DAOD_TRUTH1.e.
8338_e7400_p3401", so we apply the findDatasets (expression) function to the appropriate container name. We obtain a Spark DataFrame as a result, where we can count the number of entries to check the number of datasets included in this container (38). In addition, we can see several fields for each dataset, including:

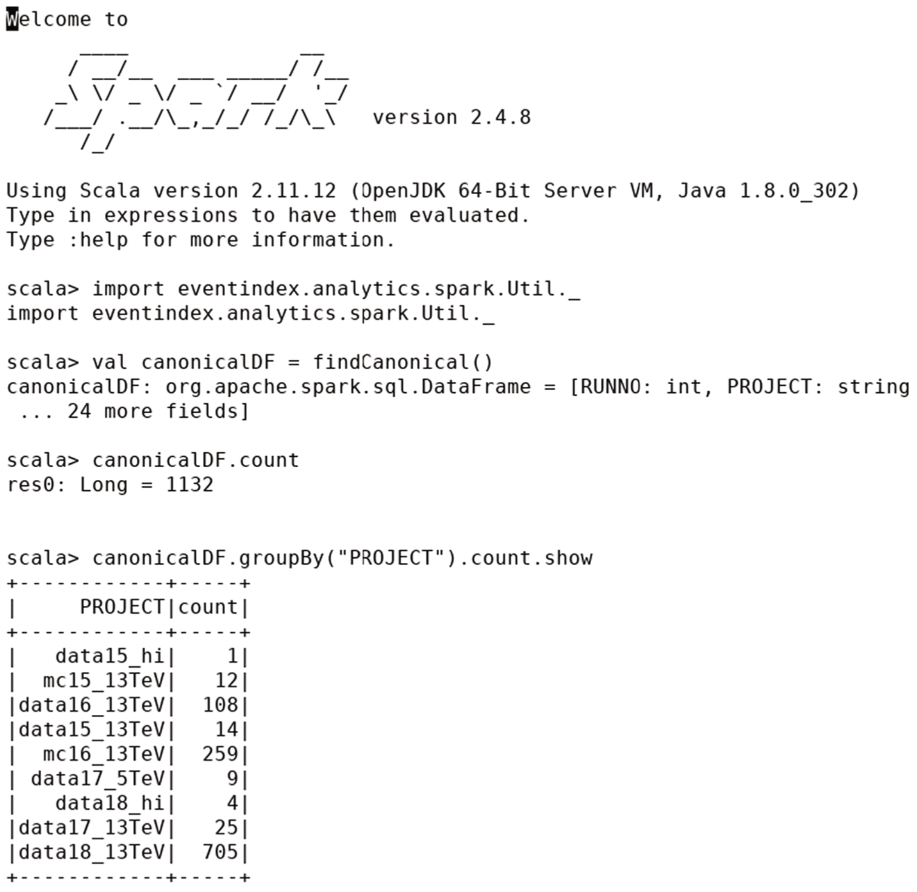
(i) The dataset name (common for all datasets in a container but the suffix "_tidnnnnnnn", where nnnnnnn is a production system task number (7 digits on left zero filling), then the dataset contains only data made by the production task nnnnnn

(ii) the numeric production system task number just mentioned (TID)

(iii) The number of files (files) and files affected with duplicates (files_dup)

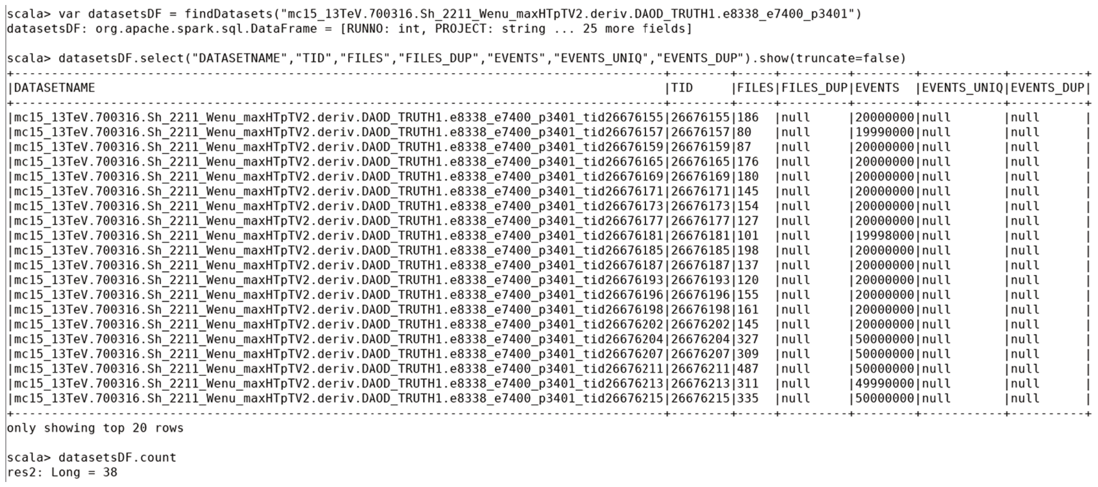
(iv) The number of unique event identifiers (events_uniq) and duplicate (events_dup) events

Therefore, the number of unique and duplicate events and the files that contain duplicates are not available at the dataset level either, so will have to be calculated with the functions defined in the following subsections.



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.8
      /_/

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_302)
Type in expressions to have them evaluated.
Type :help for more information.

scala> import eventindex.analytics.spark.Util._
import eventindex.analytics.spark.Util._

scala> val canonicalDF = findCanonical()
canonicalDF: org.apache.spark.sql.DataFrame = [RUNNO: int, PROJECT: string
 ... 24 more fields]

scala> canonicalDF.count
res0: Long = 1132


scala> canonicalDF.groupBy("PROJECT").count.show
+------------+-----+
|     PROJECT|count|
+------------+-----+
|   data15_hi|    1|
|  mc15_13TeV|   12|
|data16_13TeV|  108|
|data15_13TeV|   14|
|  mc16_13TeV|  259|
| data17_5TeV|    9|
|   data18_hi|    4|
|data17_13TeV|   25|
|data18_13TeV|  705|
+------------+-----+
```

Figure 8: Data discovery functions. DataFrame building from the canonical datasets table.

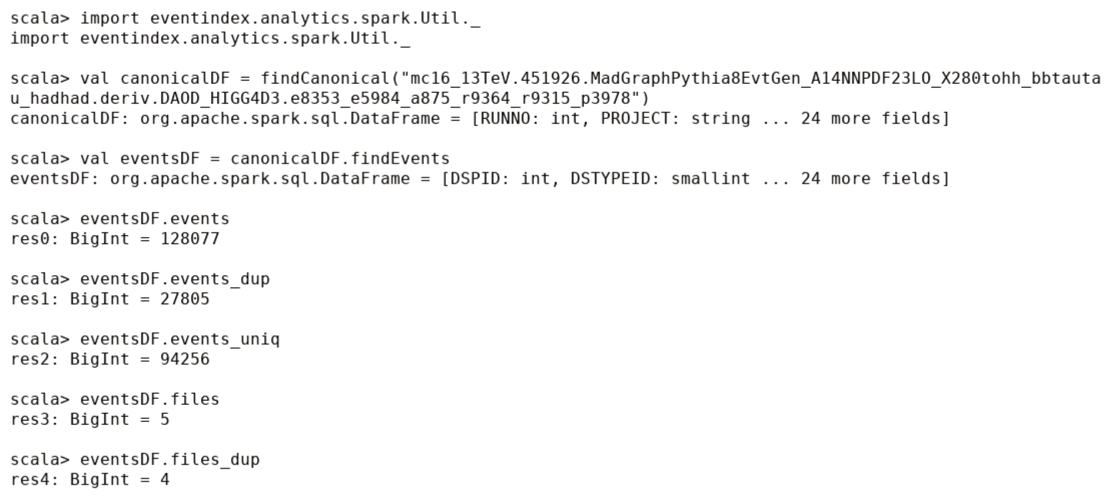


```
scala> var datasetsDF = findDatasets("mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401")
datasetsDF: org.apache.spark.sql.DataFrame = [RUNNO: int, PROJECT: string ... 25 more fields]

scala> datasetsDF.select("DATASETNAME","TID","FILES","FILES_DUP","EVENTS","EVENTS_UNIQ","EVENTS_DUP").show(truncate=false)
+----------------------------------------------------------------------------------------+--------+-----+---------+--------+-----------+----------+
|DATASETNAME                                                                             |TID     |FILES|FILES_DUP|EVENTS  |EVENTS_UNIQ|EVENTS_DUP|
+----------------------------------------------------------------------------------------+--------+-----+---------+--------+-----------+----------+
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676155|26676155|186  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676157|26676157|80   |null     |19990000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676159|26676159|87   |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676165|26676165|176  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676169|26676169|180  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676171|26676171|145  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676173|26676173|154  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676177|26676177|127  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676181|26676181|101  |null     |19998000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676185|26676185|198  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676187|26676187|137  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676193|26676193|120  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676196|26676196|155  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676198|26676198|161  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676202|26676202|145  |null     |20000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676204|26676204|327  |null     |50000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676207|26676207|309  |null     |50000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676211|26676211|487  |null     |50000000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676213|26676213|311  |null     |49990000|null       |null      |
|mc15_13TeV.700316.Sh_2211_Wenu_maxHTpTV2.deriv.DAOD_TRUTH1.e8338_e7400_p3401_tid26676215|26676215|335  |null     |50000000|null       |null      |
+----------------------------------------------------------------------------------------+--------+-----+---------+--------+-----------+----------+
only showing top 20 rows

scala> datasetsDF.count
res2: Long = 38
```

Figure 9: Data discovery functions. Search datasets with expressions.

4.2. Duplicate Calculation. There is a need to detect event duplication at different granularities, starting from files (GUID) containing duplicates but also at a higher level. The EventIndex Distributed Data Collection can detect duplicates at the file (GUID) level when indexing the input GUID files [4], so this information is conveyed and it is available in the data backend since the first ingestion. However, this is not the case on the dataset or container level, where we have to apply analytic tools to check for event duplicates.

Our tool provides a set of functions to detect duplicates at several granularities. It also expands the functionality of the Spark DataFrames containing events, providing custom transformation functions related to the calculation of duplicates.



```
scala> import eventindex.analytics.spark.Util._
import eventindex.analytics.spark.Util._

scala> val canonicalDF = findCanonical("mc16_13TeV.451926.MadGraphPythia8EvtGen_A14NNPDF23LO_X280tohh_bbtauta
u_hadhad.deriv.DAOD_HIGG4D3.e8353_e5984_a875_r9364_r9315_p3978")
canonicalDF: org.apache.spark.sql.DataFrame = [RUNNO: int, PROJECT: string ... 24 more fields]

scala> val eventsDF = canonicalDF.findEvents
eventsDF: org.apache.spark.sql.DataFrame = [DSPID: int, DSTYPEID: smallint ... 24 more fields]

scala> eventsDF.events
res0: BigInt = 128077

scala> eventsDF.events_dup
res1: BigInt = 27805

scala> eventsDF.events_uniq
res2: BigInt = 94256

scala> eventsDF.files
res3: BigInt = 5

scala> eventsDF.files_dup
res4: BigInt = 4
```

Figure 10: Duplicate event calculation. An events DataFrame implements custom transformation functions related to the calculation of duplicates.

As defined in the previous sections, the following functions are provided and can be used to calculate the missing values in the defined dataset and container tables:

(i) events(): number of event entries

(ii) events_dup(): number of events with duplication

(iii) events_uniq(): number of unique events (identifiers)

(iv) files(): number of files (GUIDs) seen

(v) files_dup(): number of files with duplicates

In Figure 10, we show an example where we apply the previous duplicate calculation function to a mc16_13 TeV canonical container dataset with the name "mc16_13TeV.451926.MadGraphPythia8EvtGen_A14NNPDF23LO_X280tohh_bbtautau_hadhad.deriv.DAOD_HIGG4D3.e8353_e5984_a875_r9364_r9315_p3978" but can be applied similarly to standard datasets.

First, we import the Spark eventindex package as usual, and then we find the canonical container and store it in the canonicalDF Spark DataFrame. Then, we apply the transformation to get the events for that canonical container, obtaining them in the eventsDF Spark DataFrame.

The subsequent functions are applied in that DataFrame to obtain the number of total events (128,077), the number of events with duplication (27,805), the number of unique event identifiers (94256), the total number of files (5), and the number of files affected with duplicates (4).

These calculated values can be stored in the related tables by the user, or by any automated higher-level tool.

4.3. Evaluation. A Spark application using these functions was implemented to be submitted automatically to the production system and to be applied with different granularities including a dataset with all datatypes (specifying only its *dspid*), a canonical dataset (additionally specifying its *dstypeid*), and only a production dataset belonging to a canonical dataset (additionally specifying its *TID*).

This program calculates the number of event records, unique event identifiers, duplicated event identifiers, the number of files that contain duplicated event identifiers (in the context of the granularity calculated), and a list of files (GUIDs) with duplicated event identifiers and the number of that event records within that file. Result of this program is a summary JSON file with these variables, with the final object of filling in the missing values in the datasets and canonical dataset container metatables.

We measured the performance of this program on the same examples, datasets (identified by dspid) with one or more derivations, corresponding to one or more datatypes. The size of these samples in the number of events varies from 200 k, 1 M, 20 M, and 100 M and contains 1 to 7 derivations (datatypes). We have chosen this set of examples as they contain duplicates, although we have also tested the base cases for samples when there are no duplicates.

The procedure measures the time from the start of the main Scala process to the end of the calculations. Therefore, it does not include the interactive or background setup times in case the application is sent to be executed in a YARN cluster, but only real data access and computation time.

A Spark application consists of a driver process that analyzes and distributes the work to a set of executor processes running in the cluster.

We submitted the same application for the same samples several times, to measure variability in cluster utilisation and other variables. The cluster configuration makes it possible to distribute the load among 4 Spark executors initially, but that automatically scales up to 32 executors.

We can see the results in Figure 11, and we observe that even with a nonexistent dspid (so 0 event sample). The procedure has to check the event table, and it takes a baseline time of 113 seconds. When applying to a 200 k dataset without any duplicates, the procedure time takes 150 seconds as a median (ranges from 117 to 216 s). When finding duplicates, a 200 k dataset time is 150 seconds as a median as well.

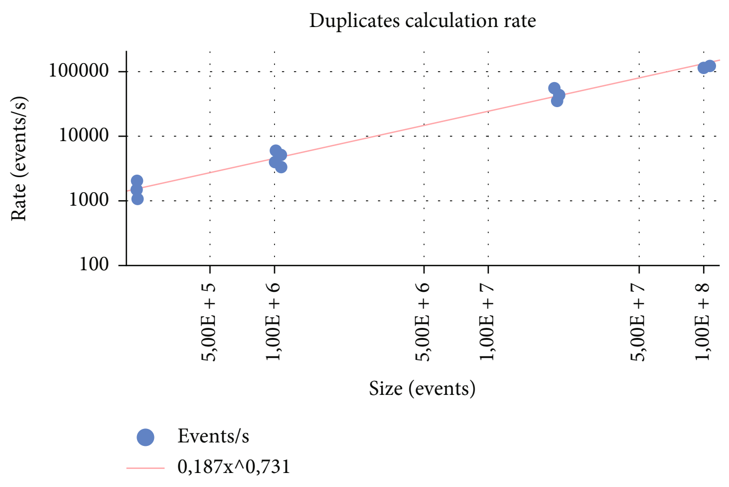


Figure 11: Duplicate events for a dataset sample calculation rate. The horizontal axis shows the size of the sample of events in a logarithmic scale. The vertical axis shows the rate of processed events per second in a logarithmic scale.

Time ranges from 112 to 171 seconds, detecting from hundreds to 24 k duplicates in 54 files. Overall, the mean rate for 200 k event datasets is a rate of 1.5 k events/s.

The 1 M event datasets take from 171 to 314 seconds (3-6 k events/s). The 20 M event datasets take from 347 to 564 seconds (35-57 k events/s). In all of these samples, the number of detected duplicates and files containing duplicates varies without determining a clear weight on the resulting processing times. Samples of the same size with more duplicated events and number of affected files can yield better results, making the size of the sample the determinant variable to predict the processing time.

Bigger datasets with their derivations comprising about 100 M events take 869-884 seconds (120 k events/s). Again, there is not much difference on the processing time attributable to the number of duplicates detected (8 to 23 million) and affected files (500 to 200 files). The load is distributed among 4 executors initially, and it scales up to 32 executors automatically, as defined by the cluster configuration.

It has to be noted that in case of a high number of affected files, the detailed list might occupy a nonnegligible amount of space (530 Mbyte JSON files with details of ~4 M events duplicated on 422 files). The time of producing the output summary file is not shown here. In particular, in this sample, it was about 60 seconds for collecting the output of the driver process and writing the file. This time can be decreased if the output is written directly to the Spark executors in a distributed manner.

4.4. Helper Functions. Our tool also provides some additional helper functions:

(i) *withColumnGUID().* adds a new DataFrame column with a decoded GUID file (from the SelfReference field)

(ii) *groupByGuid().* groups the entries by file (GUID), effectively calculating the number of event entries per file

(iii) *groupByEventno().* # entries per event number identifier

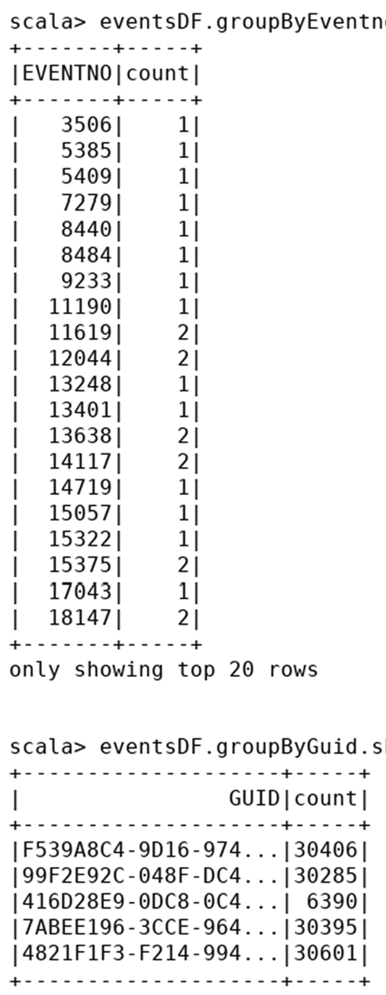
In Figure 12, we first apply the function "groupByEventno()" to the eventsDF dataframe. We obtain in the first column a list of event identifiers (event number or eventno), and in the second column, the number of entries (count) of that event identifier in the dataset. The example shows the first 20 rows, and we can observe that some event identifiers have a count of 2 (i.e., evento = 11619), so this means that this event identifier has 2 entries and therefore is a duplicated event. We could refine even more this result DataFrame, for example, by applying a "where count ≥2" clause to obtain only the event identifiers that have duplicates.

The bottom part of the figure shows the "groupByGuid()" function applied to the same dataframe, which results in the files (identified by GUID) in the first column, and the number of event entries (count) that contain each of the files in the second column.

Function *withColumnGUID()* is internally used by *groupByGuid*, but it can also be used if there is the need to obtain the GUID identifiers for other calculations or statistics, as they are not directly available in the backend tables and have to be decoded from the "self-reference" field.

Trigger is encoded with fields in the events table as an array of shorts. The Apache Phoenix Spark connector was incorrectly interpreting this data type, making access impossible from Spark and therefore from our tool. Some modifications and integration tests were developed to solve this issue in the Phoenix Spark connector, and they were submitted to the Apache Phoenix project and adopted [20].

4.5. Overlap Calculation. One of the use cases of the ATLAS EventIndex is the calculation of event overlap matrices among the derivations of a dataset. The reason is that an event is reprocessed and stored in several formats (several output files along the time). In particular, for the derivation



```
scala> eventsDF.groupByEventno.show
+-------+-----+
|EVENTNO|count|
+-------+-----+
|   3506|    1|
|   5385|    1|
|   5409|    1|
|   7279|    1|
|   8440|    1|
|   8484|    1|
|   9233|    1|
|  11190|    1|
|  11619|    2|
|  12044|    2|
|  13248|    1|
|  13401|    1|
|  13638|    2|
|  14117|    2|
|  14719|    1|
|  15057|    1|
|  15322|    1|
|  15375|    2|
|  17043|    1|
|  18147|    2|
+-------+-----+
only showing top 20 rows


scala> eventsDF.groupByGuid.show
+--------------------+-----+
|                GUID|count|
+--------------------+-----+
|F539A8C4-9D16-974...|30406|
|99F2E92C-048F-DC4...|30285|
|416D28E9-0DC8-0C4...| 6390|
|7ABEE196-3CCE-964...|30395|
|4821F1F3-F214-994...|30601|
+--------------------+-----+
```

Figure 12: Helper functions to group by event number and by file (GUID).

framework; currently, there are n streams being produced which will be spread among several trains (processing jobs) and will end in n files.

Therefore, 1 input file, n output files, the event overlap between these needs to be monitored.

We wish to determine how many and which events end up in each stream. For a number of datasets. The provided tool function calculateOverlaps() calculates the values needed to build the matrix for all derivations of a given data-set with its identifier.

Figure 13 shows the execution of the function over the DAODs derived from the AOD dataset: “data18_13TeV.00350144.physics_Main.merge.AOD.f933_m1960”.

For reference, DAOD (Derived AOD) datasets have the following names:

data18_13TeV:data18_13TeV.00350144.physics_Main.-deriv.DAOD_BPHY1.f933_m1960_p3553.

data18_13TeV:data18_13TeV.00350144.physics_Main.-deriv.DAOD_BPHY4.f933_m1960_p3553.

data18_13TeV:data18_13TeV.00350144.physics_Main.-deriv.DAOD_BPHY5.f933_m1960_p3553.

…

They comprise 83 datasets summing up around 500 M events. The execution of our algorithm results in a Spark DataFrame that can be shown in Figure 13.

The result dataframe shows an entry for every pair of derived streams that contain overlapped events. Therefore, for N-derived streams, we would have an N X N matrix, but we have to bear in mind that the events overlapping in a pair of streams (i,j) will hold the same results as the pair of streams (j,i), so we will have a symmetric matrix. In addition, the elements of the leading diagonal (i,i) that contains the values of one stream against itself will be always the same (ratio = 1 as all events are by definition the same events_stream1_only = events_stream2_only = 0, and events_both-streams will equal the total number of events in the stream). Therefore, instead of $N \times N$ elements, we will have to explicitly calculate only the $n(n-1)/2$ elements in the upper right (or lower left), which are the independent entries of the matrix.

In this example, the results on the screen show the first 20 rows or entries of the result overlapsDF DataFrame that contains $83 * (83-1)/2 = 3403$ entries.

As an example, if we take the first entry, the (EXOT2, TAUP1) streams will produce the same results as (TAUP1, EXOT2) as the overlapped events in both streams are the same, and so the rest of the values. In this case, we see that there are 13349380 events that are only in EXOT2 (events_stream1_only) and 3320739 events in TAUP1 (events_stream2_only). Then there are 99177 events that are in both streams, so this is the number of overlapped events or the intersection of both streams. We calculate the ratio (0.005914…) which represents the events that are in both streams, to the sum of total events. This is calculated as the intersection (events_both_stream) over the union (events_stream1_only+events_stream2_only+events_both_streams).

This DataFrame can as well be stored in an output file, or in another Phoenix Table, like in the following example. We have stored the results in another table DATASETS_OVERLAPS with the same schema as the overlaps DataFrame.

We have previously seen in our data model that millions of events reside in a big events table, with a row per event entry. We will apply the algorithm only to the needed data, namely, the event entries stored for every derived dataset that we are taking into account.

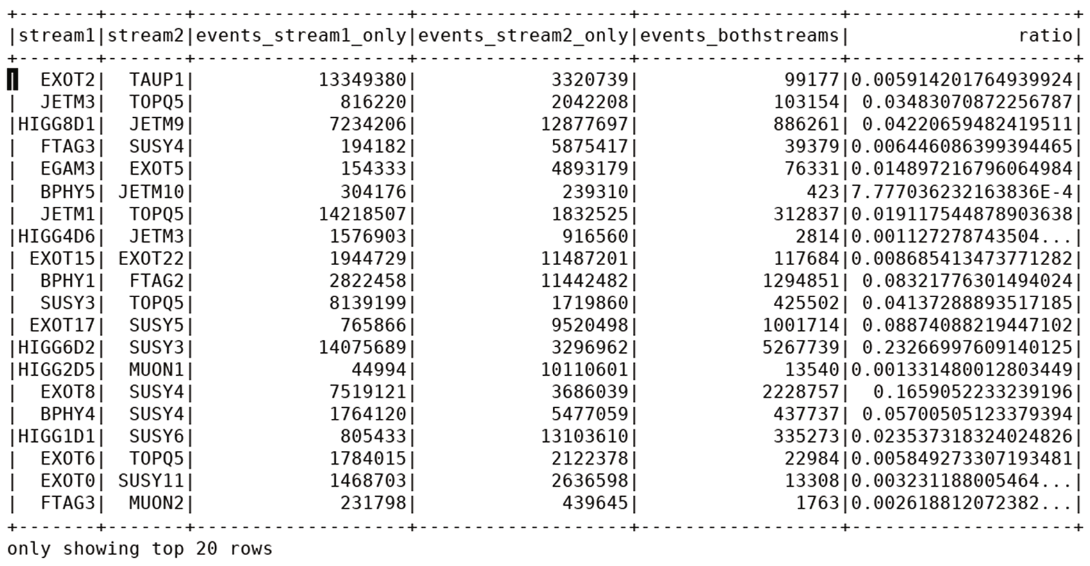
In Algorithm 1, we show the pseudocode for the overlap calculation algorithm that has 4 main steps:

Step 1. For every event record, select the event identifier (eventnumber), and the stream (datatype).

(a) The result of this step is a set of (EventId, stream). This set might contain several entries with the same Event Id

Step 2. Group the streams by the event identifier

(a) Result of this step is a set of (EventId, EventStreams), where EventStreams is a set (Stream1, Stream2, … StreamN). The number of elements of this set of



```
+-------+-------+------------------+------------------+-----------------+--------------------+
|stream1|stream2|events_stream1_only|events_stream2_only|events_bothstreams|               ratio|
+-------+-------+------------------+------------------+-----------------+--------------------+
|  EXOT2|  TAUP1|          13349380|           3320739|            99177|0.005914201764939924|
|  JETM3|  TOPQ5|            816220|           2042208|           103154| 0.03483070872256787|
|HIGG8D1|  JETM9|           7234206|          12877697|           886261| 0.04220659482419511|
|  FTAG3|  SUSY4|            194182|           5875417|            39379|0.006446086399394465|
|  EGAM3|  EXOT5|            154333|           4893179|            76331|0.014897216796064984|
|  BPHY5| JETM10|            304176|            239310|              423|7.777036232163836E-4|
|  JETM1|  TOPQ5|          14218507|           1832525|           312837|0.019117544878903638|
|HIGG4D6|  JETM3|           1576903|            916560|             2814|0.001127278743504...|
| EXOT15| EXOT22|           1944729|          11487201|           117684|0.008685413473771282|
|  BPHY1|  FTAG2|           2822458|          11442482|          1294851| 0.08321776301494024|
|  SUSY3|  TOPQ5|           8139199|           1719860|           425502| 0.04137288893517185|
| EXOT17|  SUSY5|            765866|           9520498|          1001714| 0.08874088219447102|
|HIGG6D2|  SUSY3|          14075689|           3296962|          5267739| 0.23266997609140125|
|HIGG2D5|  MUON1|             44994|          10110601|            13540|0.001331480012803449|
|  EXOT8|  SUSY4|           7519121|           3686039|          2228757|  0.1659052233239196|
|  BPHY4|  SUSY4|           1764120|           5477059|           437737| 0.05700505123379394|
|HIGG1D1|  SUSY6|            805433|          13103610|           335273|0.023537318324024826|
|  EXOT6|  TOPQ5|           1784015|           2122378|            22984|0.005849273307193481|
|  EXOT0| SUSY11|           1468703|           2636598|            13308|0.003231188005464...|
|  FTAG3|  MUON2|            231798|            439645|             1763|0.002618812072382...|
+-------+-------+------------------+------------------+-----------------+--------------------+
only showing top 20 rows
```

Figure 13: Overlap event calculation for a set of 83 datasets with 500 M events.

```
// Step 1
method map1 (events)
    for all x in events do
        emit(x.EventId, x.Stream)
// Step 2
method reduce1 ( eventId, Stream [s1, s2, ...])
    EventStreams <= new AssociativeArray
    for all stream in [s1, s2, ...] do
        EventStreams <= stream
    emit (EventId, EventStreams)
// Step 3
AllStreamsList <= new List(s1, s2, .., sn)
method map2 (EventId, EventStreams)
    numStreams = length(AllStreamsList)
    for i in 1 to numStreams do
        for j in i+1 to numStreams do
            streamI <= AllStreamsList[i]
            streamJ <= AllStreamsList[j]
            isInI <= true if EventStreams{streamI} exists
            isInJ <= true if EventStreams{streamJ} exists
            if isInI or isInJ do
                emit (pair(streamI, streamJ), pair(isInI, isInJ))
// Step 4
method reduce2 (Pair(StreamI, StreamJ), Pair(isInI, isInJ) [p1, p2, ...])
    events_stream1_only <= 0
    events_stream2_only <= 0
    events_both_streams <= 0
    ratio <= 0
    for all p(i, j) in [p1, p2, ...] do
        case (true, false) : events_stream1_only <= events_stream1_only + 1
        case (false, true) : events_stream2_only <= events_stream2_only + 1
        case (true, true) : events_both_streams <= events_both + 1
    ratio <= events_both_streams /
        (events_stream1_only + events_stream2_only + events_both_streams)
    emit (pair(StreamI, StreamJ), events_stream1_only,
        events_stream2_only, events_both_streams, ratio))
```

Algorithm 1: Overlap calculation algorithm pseudocode.

streams corresponds to the number of event entries of a particular event identifier

Step 3. For every event identifier, build all pairs of streams (i,j) that might contain this particular event, signalling where it is found

(a) Result of this step is a set of tuples [(StreamI, StreamJ), (is_in_I, is_in_J)], where is_in_X is a boolean that signals that this event entry is found in that stream. Values emitted might be:

   (1) (false, false): not found in any, so this value is not emitted at all and will not be found in the set of tuples

   (2) (true, false): the event entry is found in StreamI but not in StreamJ

   (3) (false, true): the event entry is found in StreamJ but not in StreamI

   (4) (true, true): the event entry is found both in StreamI and StreamJ, so this is an overlap

(b) It has to be noted that we have to build pairs of streams not only from the EventStreams set in Step 2 (as will contain only the overlaps) but to travel all possible pairs of streams (i,j). With s streams, this means as much as $(s * (s-1))/2$ entries. This counts the events that might be in one but not in the other stream. As stated, if both i and j streams are not found in the EventStreams set considered in this step, then the (false,false) value is not emitted, reducing the potential $(s * (s-1))/2$ values emitted per entry generated from the previous step

Step 4. Group the tuples by pairs of streams, counting the number of previously generated values

(a) Result is a set of Tuples [(StreamI, StreamJ), (events_stream1_only, events_stream2_only, events_both-streams, ratio)]

(b) The set contains an entry per (StreamI, StreamJ) pair possibility, with as many as $(s * (s-1))/2$ entries

(c) When grouping by pair (StreamI, StreamJ), the values (is_in_I, is_in_J) previously emitted are counted in the mentioned variables:

   (1) (true, false) sum 1 to events_stream1_only

   (2) (false, true) sum 1 to events_stream2_only

   (3) (true, true) sum 1 to events_both_streams

   (4) ratio is calculated as the intersection over the union, so ratio = events_both_streams/(events_stream1_only + events_stream2_only + events_both_streams)

The result of Step 4 is what we find in the output of the example shown before, the set of unique entries of the matrix that represents the possible $S \times S$ overlaps of the S-derived streams of the analyzed dataset.

The implementation of the algorithm is done in Scala language and using Spark abstractions and functions. First, we are reading the events from the source (Events HBase tables with Phoenix Schema) that are from the streams of interest, in this case, all streams (dstypeid) found in the canonical container table for a particular dataset by its dataset identifier (dspid).

Since all data share the dspid which is the key prefix (see Section 3.1), we assure the locality of the data. In addition, all events from a stream datatype will share the dstypeid, that is, the next data in the key prefix, so they will be together in disk and not spread, also assuring the locality of the data.

Step 1 of the algorithm is achieved with a map() transformation, retrieving only the fields of interest of every event entry, namely, the event identifier (eventnumber or eventno) and the stream (dstypeid).

In Step 2, we apply the aggregateByKey() transformation which is much more efficient in the Spark Scala implementation as it can be applied in parallel in different partitions where the data is instead of moving the data as in groupByKey(). We want the result of the aggregation to be a set of values, that is a different type than the values that are strings (the sum of strings is a concatenation of the string), so we use this function instead of reduceByKey().

The backend data is organized into all streams (dstypeid) of a dataset (dspid) to be consecutive in the HBase row key space and therefore in the storage (disks). We can benefit that most of these calculations are done in the same machine and in memory in particular, without too much data shuffling across the region servers of the cluster. It is, however, possible that for big datasets and lots of derivation streams, their data expands along several region servers. In this case, the aggregateByKey() transformation will shuffle the data to aggregate by the event identifier.

In Step 3, we use the flatMap() transformation to the DataFrame result of Step 2 which has an entry per event identifier, into the set of tuples that identify pairs of streams and where in that pair the event entry is found (as boolean pairs described in the algorithm). This flatMap() transformation applies a custom eventsStreamsPairMapper() function to every original entry (event identifier) for that purpose.

The last Step 4 applies again an aggregateByKey() transformation to reduce the previous results by pairs of streams. The first parameter of aggregateByKey() will be the combiner function for merging values within a partition, taking the boolean pairs, and converting them to tuples (events_stream1_only, events_stream2_only, events_both_streams, ratio). Ratio is not computed in this combiner function, yet it is emitted as 0. Yet the summatories of the events are done at the partition level.

The second parameter of aggregateByKey() will be the reducer function to group a pair of tuples produced by the

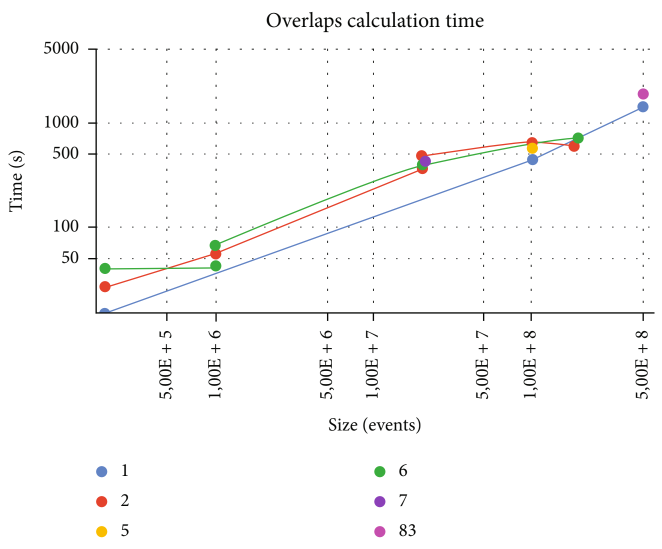


Figure 14: Overlap calculation time. Every line represents the number of streams (s) of the problem. Horizontal axis represents the size of the problem in events in logarithmic scale; the vertical axis represents the time in seconds, in logarithmic scale.

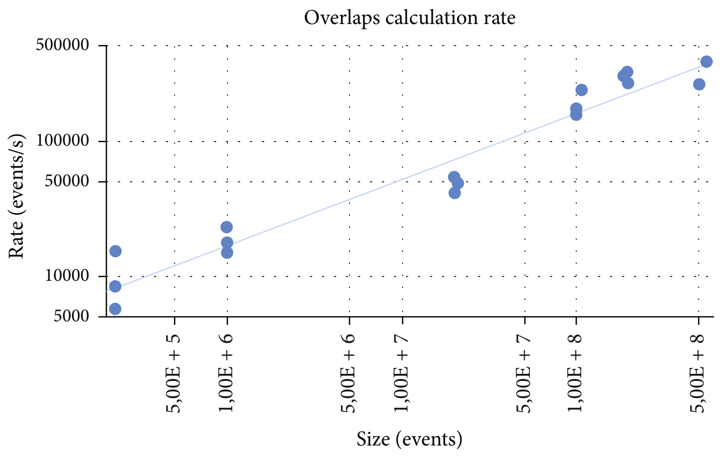


Figure 15: Overlap calculation rate. Horizontal axis represents the size of the problem in events in logarithmic scale; the vertical axis represents the rate of events processed per second, in logarithmic scale.

previous function and applied when merging values between partitions. In this case, the output is another tuple summing the 3 first values and computing the ratio as the intersection over the union (events_both_streams/(events_stream1_only + events_stream2_only + events_both_streams)).

Due to a feature in the Spark Scala implementation, this reducer function is not applied when all data is in a single partition, so the ratio will not be computed correctly. In this case, there will be another step to explicitly compute the ratio when producing the final results.

The last part of the implementation deals with showing user-friendly values in the result DataFrame, which implies converting the dstypeid to the user-friendly stream names stored in the dstypes tables.

4.6. Evaluation. We have tested the algorithm on several datasets and derivation streams. The most common data currently has few stream derivations ($s < 10$), with datasets with size n that range from thousands to millions of events. Only one dataset sample contains 83 derivations.

The sizes of the problem (n) of the dataset of the samples tested are 200 k, 1 M, 20 M, 100 M, and 500 M. These events are divided in a number (s) of streams, which are 1, 2, 5, 6, 7, and83 (unique case).

Figure 14 shows the overlap calculation time depending on the size of the dataset on the x-axis and the number of streams (in lines with different colours).

For dataset samples up to 200 k events, the overlap procedure cost from 15 seconds for a dataset with just 1 stream. Therefore, this is the baseline as no matrix elements are calculated. Then, it takes 27 seconds for 2 streams and 40 seconds for 6 streams.

For dataset samples of 1 M events, it takes from 43 to 67 seconds. A 6 stream sample takes 43 seconds, the same as a 200 k dataset, so in this case this might be an issue in the former calculations.

For bigger datasets, they rise almost linearly, and there are only a few samples in our system for datasets bigger than 10^8, but it is worth testing as these cases are interesting for the final users. There is only one sample with 83 streams and 500 M events that has multiple overlaps.

In general, series converge to the baseline case with 1 stream when there are no overlaps showing that the main factor affecting performance is the dataset size.

The number of streams affects how Steps 2–4 of the algorithms are applied, increasing the number of temporary results and affecting how many comparisons are to be made. The number of computations performed is $n * ((s * (s-1))/2)$, where n is the number of events and s is the number of streams.

As we can see also in Figure 14, the cost is dominated by the size of the problem n, while the cost of computing the $(s * (s-1))/2$ pairs per entry takes smaller time compared with the n term.

A quadratic term dominates a linear one. Nevertheless, in this case, the quadratic one is of much less magnitude, so in the algorithm cost, the dominant term is n, and the temporal cost is $O(n)$.

The intermediate data produced is at most the number of computations $n * ((s * (s-1))/2)$ in Step 3 of the algorithm presented. However, this data is constantly reduced per spark partition at Step 4. Thus, the final space cost is $O(s^2)$.

Figure 15 shows the overlap calculation rate. For datasets of 20 M events, processing takes 368-483 seconds and yields a processing rate of about 50 k events/s. Again, datasets with higher number of streams might yield better results, revealing again the preponderance of the (event) size factor.

100 M event baseline dataset with just 1 stream takes 445 seconds and yields a performance of 240 k events/s processed. Then datasets with more streams and overlap processing take 579-643 seconds, with a mean 160 k events/s processed. The biggest 500 million events sample yields a performance of 380 k events/s.

5. Conclusions

We have presented a framework and a set of analytical tools using Spark abstractions and implemented in a Scala package. With it, we are accessing billions of event records in an HBase data backend. An Apache Phoenix layer provides schema enforcement and SQL capabilities to access the data. With our tools, we abstract the backend data model, decoupling the data access from the actual data schema and used technologies. This approach is very convenient to hide model complexity with an accessible defined interface. It also masks changes in the data model, which are invisible for the user as the defined interfaces do not change.

The package can be used interactively within a command-line spark-shell session. It can also be used with batch standalone Spark jobs, as we have shown when evaluating our tool algorithms.

We have shown that our data model is defined in a big events table with data organized in 4 families, and other metatables for data discovery and bookkeeping. For the evaluation, we have used dataset samples of EventIndex production data that were already available in HDFS, and we have imported it in our HBase/Phoenix system. We have used a MapReduce approach bulk importing campaign that allowed us to ingest 70 billion events from almost 8,000 datasets in a week, with a mean rate of 117 kHz (events/s). Eventually, our event table occupied 22 TB in the CERN HBase cluster.

The tool and algorithms presented solve data access for our application use cases in areas like data discovery, duplicate detection, and overlap calculation. Data discovery capabilities produce Spark DataFrames usable by the rest of the tools. They are also using custom helper functions to access the encoded fields of the data model. During the process of development, some limitations in the Apache Phoenix Spark connector were solved and contributed back to the community.

An overlap calculation algorithm was presented with the computational cost $O(n)$ with the number of events and spatial cost $O(s^2)$ with the number of streams. Implemented in Scala and using Spark abstractions, it translates automatically to scans over the HBase key, which is fast and performant. All data for the derivation of a dataset are adjacent in the key space, and therefore storage, reducing input/output operations. The algorithm scales automatically within the Spark cluster up to 32 processes, yielding a performance of 380 k events processed per second for a 500 M event dataset.

The duplication detection case accesses the event table key and family A (event location) data. It is slower compared with the overlap case due to the access of different values in a data family and not only the HBase key, yielding a performance of 120 k events per second for 100 M events, compared with 150 k events/second for the same size overlap calculation.

The processing rates show a penalization for smaller datasets that are due to the setup of HBase data streams, dominating the accounted time. We obtain better rates for bigger dataset sizes, so one possibility is to increment spark job granularity when possible, for example, calculating features at the container level, instead of its constituent datasets.

The presented framework approach solves the analytic use cases of the ATLAS EventIndex project in a performant manner, providing convenient data access paths which will be exploited starting with the LHC Run 3 (2022-2025).

Data Availability

The data utilized to support the findings of this analysis are available from the corresponding author upon request.

Disclosure

This work was first presented in CMMSE22, the International Conference on Computational and Mathematical Methods in Science and Engineering, and was granted the prize for "Best computational applications on line presentation" [21, 22].

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This work has been partially supported by the Spanish Ministerio de Ciencia e Investigación under project PID2019-104301RB-C21.

References

[1] L. Evans and P. Bryant, "LHC machine," *Journal of Instrumentation*, vol. 3, no. 8, article S08001, 2008.

[2] Atlas Collaboration, "The ATLAS experiment at the CERN Large Hadron Collider," *Journal of Instrumentation*, vol. 3, no. 8, 2008.

[3] D. Barberis, J. Cranshaw, G. Dimitrov et al., "The ATLAS EventIndex: an event catalogue for experiments collecting large amounts of data," *InJournal of Physics: Conference Series*, vol. 513, no. 4, article 042002, 2014.

[4] D. Barberis, I. Alexandrov, E. Alexandrov et al., "The ATLAS EventIndex," *Computing and Software for Big Science*, vol. 7, no. 1, p. 2, 2023.

[5] "Apache HBase," http://hbase.apache.org.

[6] "Apache Phoenix," https://phoenix.apache.org/.

[7] M. Zaharia, R. S. Xin, P. Wendell et al., "Apache Spark," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[8] HSF Physics Event Generator WG, A. Valassi, E. Yazgan et al., "Challenges in Monte Carlo event generator software for hyigh-luminosity LHC," *Computing and Software for Big Science*, vol. 5, no. 1, p. 12, 2021.

[9] J. Catmore, J. Cranshaw, T. Gillam et al., "A new petabyte-scale data derivation framework for ATLAS," *Journal of Physics: Conference Series*, vol. 664, no. 7, article 072007, 2015.

[10] M. V. Perez, E. Alexandrov, I. Aleksandrov et al., "The ATLAS EventIndex and its evolution towards Run 3," *In Journal of Physics: Conference Series*, vol. 1525, no. 1, article 012056, 2020.

[11] V. K. Vavilapalli, A. C. Murthy, C. Douglas et al., "Apache Hadoop YARN: yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13)*, New York, NY, USA, 2013.

[12] E. Cherepanova, S. Martinez, F. Javier et al., "The ATLAS EventIndex using the Hbase/Phoenix storage solution," in *Proceedings of the 9th International Conference "Distributed Computing and Grid Technologies in Science and Education" (GRID'2021)*, Dubna, Russia, 2021http://ceur-ws.org/Vol-3041/17-25-paper-3.pdf.

[13] S. Albrand, J. Chapman, D. Cote, L. Fiorini, and E. J. Gallas, "ATLAS dataset nomenclature," Internal Report https://cds.cern.ch/record/1070318.

[14] The ATLAS collaboration, "Operation of the ATLAS trigger system in run2," *Journal of Instrumentation*, vol. 15, no. 10, article P10004, 2020.

[15] P. Leach, M. Mealling, and R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace*, RFC 4122, 2005.

[16] E. H. Mckinney, "Generalized birthday problem," *The American Mathematical Monthly*, vol. 73, no. 4, pp. 385–387, 1966.

[17] Á. Casaní, "ATLAS EventIndex general dataflow and monitoring infrastructure," *Journal of Physics: Conference Series*, vol. 898, article 062010, 2017.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, USA, 2010.

[19] J. Dean and S. Ghemawat, "MapReduce," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] "Apache Phoenix ticket 6559," https://issues.apache.org/jira/browse/PHOENIX-6559.

[21] C. Fernandez, A. García Montoro, S. González dela Hoz, J. Salt, J. Sánchez, and V. P. Miguel, "Big Data analytics for the ATLAS EventIndex project with Apache Spark," in *Presented at CMMSE22, the International Conference on Computational and Mathematical Methods in Science and Engineering*, Cadiz, Spain, 2022.

[22] A. Fernández Casani, *Performance Improvements of EventIndex Distributed System at CERN, [Ph.D. Thesis]*, University of Valencia, 2023, https://roderic.uv.es/handle/10550/85724.