BRIEF REPORT

FunTuple: A New N-tuple Component for Offline Data Processing at the LHCb Experiment

Abhijit Mathad[1,2] · Martina Ferrillo[1] · Sacha Barré[2,3] · Patrick Koppenburg[4] · Patrick Owen[1] · Gerhard Raven[4,5] · Eduardo Rodrigues[6] · Nicola Serra[1]

Abstract

The offline software framework of the LHCb experiment has undergone a significant overhaul to tackle the data processing challenges that will arise in the upcoming Run 3 and Run 4 of the Large Hadron Collider. This paper introduces `FunTuple`, a novel component developed for offline data processing within the LHCb experiment. This component enables the computation and storage of a diverse range of observables for both reconstructed and simulated events by leveraging on the tools initially developed for the trigger system. This feature is crucial for ensuring consistency between trigger-computed and offline-analysed observables. The component and its tool suite offer users flexibility to customise stored observables, and its reliability is validated through a full-coverage set of rigorous unit tests. This paper comprehensively explores `FunTuple`'s design, interface, interaction with other algorithms, and its role in facilitating offline data processing for the LHCb experiment for the next decade and beyond.

Keywords High-energy-physics · LHCb experiment · Data processing and offline analysis

Introduction

The LHCb experiment, located at Point 8 of the Large Hadron Collider (LHC) [1] at CERN, is a forward-arm spectrometer designed to study the decays of beauty and charm hadrons [2, 3]. In the initial two runs of the LHC, during 2010–2018, the experiment (mainly) collected proton–proton collision data corresponding to a total integrated luminosity of $9\,fb^{-1}$. As preparations intensify for Run 3,[1] where the LHC's instantaneous luminosity is anticipated to surge by a factor of 5 compared to the preceding runs, the LHCb experiment is poised to enhance its capabilities even further. The upgraded detector [4] and data acquisition system will allow for improved vertexing and trigger efficiency [5]. This enhancement facilitates the exploration of exceedingly rare decays [6] while also facilitating the probing of deviations from Standard Model predictions with unparalleled precision [7–9].

The advent of Run 3 data acquisition presents significant hurdles for the LHCb data processing framework. Notably, the data volume from LHCb 's Run 3 is projected to surge by over 15 times compared to prior runs [10]. Consequently, management of petabytes of processed data and effectively incorporating distributed computing resources present significant challenges [11, 12]. In light of these challenges, a comprehensive redesign of both the trigger and offline data processing pipelines is imperative [10, 11]. This paper concentrates on the offline data processing pipeline, specifically highlighting the development of a new component called `FunTuple` [13] facilitating analysis of Run 3 data and beyond.

✉ Abhijit Mathad
  amathad@cern.ch

1 University of Zürich, Zürich, Switzerland

2 European Organization for Nuclear Research (CERN), Geneva, Switzerland

3 The University of Manchester, Manchester, UK

4 Nikhef National Institute for Subatomic Physics, Amsterdam, Netherlands

5 VU University Amsterdam, Amsterdam, Netherlands

6 Oliver Lodge Laboratory, University of Liverpool, Liverpool, UK

---

[1] The data collection from Run 3 of LHC is currently ongoing; however, the core developments emphasised in this paper transpired prior to its commencement.
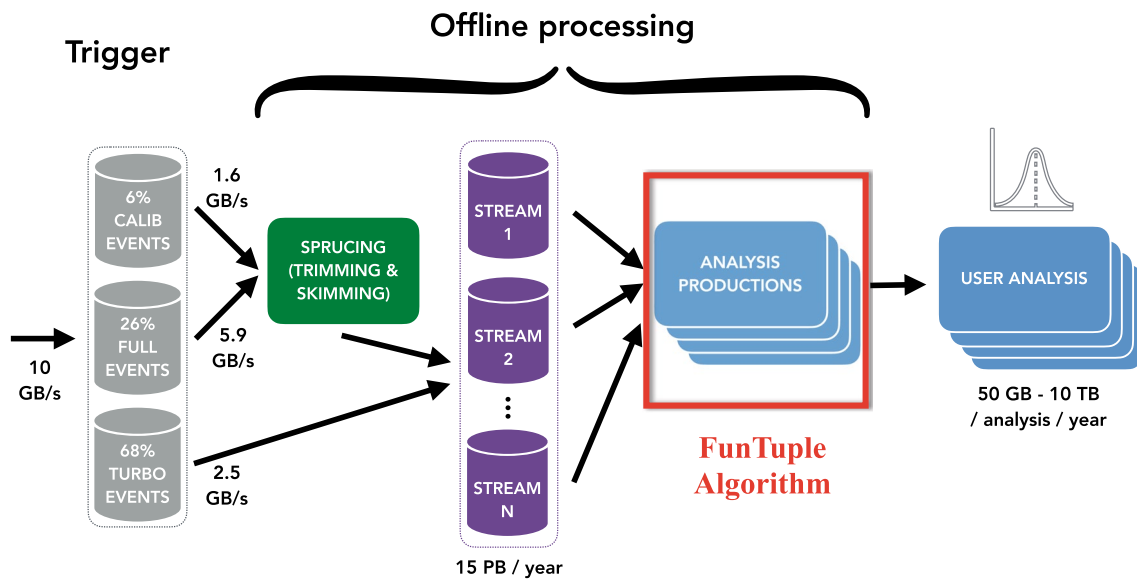
Fig. 1 Data flow diagram for Run 3 data processing showing the placement of the `FunTuple` component. Figure adapted from Ref. [30]

In the initial LHC runs, LHCb 's trigger and offline reconstruction applications, `Moore` [14] and `Brunel` [15], operated independently from the `DaVinci` application [16] employed for offline data processing. Besides executing offline event selection, the `DaVinci` application was used to process and store data for subsequent analysis. This task was accomplished via the `DecayTreeTuple` algorithm [17],[2] which recorded a specific set of observables into output files. First, due to the segregation of trigger and offline frameworks, the equivalence between trigger-computed observables and those analysed offline was not guaranteed. Second, users lacked the flexibility to customise the set of observables recorded, which is essential in light of the anticipated data volume surge for Run 3 and Run 4. Furthermore, as part of its strategy to tackle the forthcoming data processing challenges in Run 4 and beyond, the LHCb experiment plans to implement a new event model based on *Structure of Arrays* (SoA), which will facilitate vectorised processing of data [20]. Substantial enhancements were also made to the trigger reconstruction algorithms that facilitated retirement of the `Brunel` package, which was responsible for offline reconstruction [21–24]. Consequently, the development of new offline algorithms becomes imperative to accommodate these changes.

To overcome these hurdles, a strategic choice was made to leverage tools developed for the trigger system within the offline software framework. This led to the development of a new component, `FunTuple` [13], short for *Fun*ctional

n*Tuple*, which is tailored for processing Run 3 and Run 4 data. The `FunTuple` component introduces enhancements to the previous workflow. First, it guarantees the consistency between trigger-computed observables and those subjected to offline analysis. Second, FunTuple, along with all its dependencies, is entirely templated in C++, allowing it to support both legacy and upcoming event models planned for future LHC runs. The templated design along with the SoA event model enables the component to leverage *Single Instruction Multiple Data* (SIMD) vectorisation. Last, it offers users the flexibility to efficiently tailor the list of recorded observables, an important feature given the expected surge in data volume for Run 3 and Run 4. This component is configured with a robust suite of tools designed for the second stage of the LHCb trigger system, known as Throughput Oriented (`ThOr`) functors [25–27]. These functors are designed to deliver high-speed in the trigger's demanding throughput environment and are adept at computing topological and kinematic observables. `FunTuple` utilises these functors to compute a diverse range of observables and writes a `TTree` in the `ROOT` N-tuple format.[3] The N-tuple format is widely used in the High Energy Physics community to store flattened data in a tabular format [29]. Furthermore, the component's lightweight design ensures simplified maintenance and seamless knowledge transfer. As depicted in Fig. 1, the `FunTuple` component plays a central role, bridging the gap between the offline

---

[2] There were also alternative Python-based algorithms like `Bender` [18, 19] for Run 1/2 data processing.

[3] There are plans in the future to write `ROOT RNTuple`, which has been designed to address performance bottlenecks and shortcomings of `ROOT` current state of the art `TTree` [28].
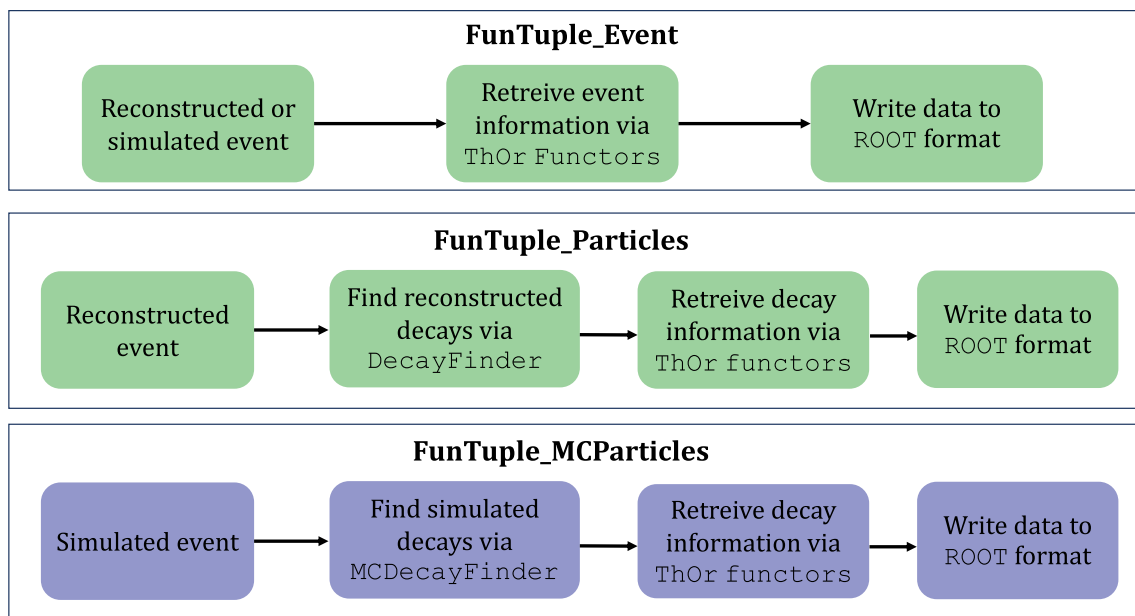
FunTuple_Event

Reconstructed or simulated event → Retreive event information via `ThOr Functors` → Write data to `ROOT` format

FunTuple_Particles

Reconstructed event → Find reconstructed decays via `DecayFinder` → Retreive decay information via `ThOr functors` → Write data to `ROOT` format

FunTuple_MCParticles

Simulated event → Find simulated decays via `MCDecayFinder` → Retreive decay information via `ThOr functors` → Write data to `ROOT` format

Fig. 2 Data flow diagram of the three flavours of `FunTuple` component

data processing stage (`Sprucing`) and the subsequent user analysis stages [30]. In the `Sprucing` stage, the data are slimmed and skimmed before being saved to disk as part of the offline data processing workflow. The placement of `FunTuple` underscores its critical role in LHCb 's analysis productions [31], facilitating the storage of experiment-acquired data in a format suitable for subsequent offline analysis.

Design and Interface

`FunTuple` is a novel component integral to the LHCb experiment's data processing infrastructure. It is a `c++` [32] class built upon the `Gaudi` functional framework [33], and it offers a user-friendly `Python` [34] interface. The flexibility of the `FunTuple` component lies in its templated design, allowing it to accommodate various types of input data. As a result, for Run 3, it is available in the three distinct flavours `FunTuple_Particles`, `FunTuple_MCParticles` and `FunTuple_Event` hereafter described.

The `FunTuple_Event` component processes input data comprising reconstructed or simulated events, where each event represents a single LHC bunch crossing. It acquires event-level information (for example, the number of charged particles in the event), using thread-safe `ThOr` functors that are specialised `c++` classes developed for utilisation in the second stage of the LHCb trigger system [25, 26, 35]. The component then stores this extracted information from `ThOr` functors in an `ROOT` N-tuple file. The `FunTuple_Particles` component functions on reconstructed

events and identifies specific reconstructed decays by utilising the decay-finding tool `DecayFinder` [27] explained in Sect. 2.1. It further retrieves essential details regarding parent and children particles (for example magnitude of the transverse moment) through `ThOr` functors and records this information in an ROOT file. Similarly, the `FunTuple_MCParticles` component shares similarities with `FunTuple_Particles`, but it processes simulated events instead, and captures information about simulated decays. For an illustrative representation of the data flow encompassing these three approaches, refer to Fig. 2. Each

```python
1  # import FunTuple to run over
       reconstructed particles
2  from FunTuple import
       FunTuple_Particles
3
4  # define instance of FunTuple
5  data_tuple = FunTuple_Particles(
6      name="TDirectoryName",
7      tuple_name="TTreeName",
8      fields=fields,
9      variables=variables,
10     event_variables=event_variables,
11     inputs=reco_data_TES_location)
```

Listing 1 `FunTuple_Particles` instance

```
1  # import FunTuple to run over
       simulated particles
2  from FunTuple import
       FunTuple_MCParticles
3
4  # define instance of FunTuple
5  data_tuple = FunTuple_MCParticles(
6      name="TDirectoryName",
7      tuple_name="TTreeName",
8      fields=fields,
9      variables=variables,
10     event_variables=event_variables,
11     inputs=mc_data_TES_location)
```

Listing 2 `FunTuple_MCParticles` instance

```
1  # import FunTuple to run over
       reconstructed or simulated event
2  from FunTuple import FunTuple_Event
3
4  # define instance of FunTuple
5  data_tuple = FunTuple_Event(
6      name="TDirectoryName",
7      tuple_name="TTreeName",
8      variables=event_variables)
```

Listing 3 `FunTuple_Event` instance

aspect of the data-flow diagram is further elaborated in the following sections.

The instantiation of the three flavours of the `FunTuple` component in `Python` is exemplified in Listings 1, 2, 3. As depicted, the user is required to provide the `name` and `tuple_name` attributes for all three flavours. The `name` attribute defines the component's name and the name of the corresponding `TDirectory` in the output ROOT file. On the other hand, the `tuple_name` attribute defines the name of the `TTree` in the ROOT file. The `fields` attribute can only be defined for `FunTuple_Particles` and `FunTuple_MCParticles` and is used to select specific decays within an event and define the corresponding `TBranches` in the output file. For a detailed exploration of this attribute, see Sect. "Finding Decays in an Event". The `variables` attribute is used to specify the observables to be computed for each event or decay. In the case of `FunTuple_Event`, only event-level observables can be defined. Conversely, for

`FunTuple_Particles` and `FunTuple_MCParticles`, both decay-level and event-level observables can be specified. The latter is achieved by defining an optional `event_variables` attribute. It is worth noting that the `FunTuple` component automatically writes certain event information, such as the run and event numbers,[4] to the output file by default. For a more comprehensive discussion on the `variables` attribute, refer to Sect. "Retrieve Event and Decay Information". Finally, the `inputs` attribute refers to the Transient Event Store (TES) location, indicating the data pertaining to a given event cycle that will be processed. Subsequently, the processed information is stored in the output ROOT file, which is further elaborated on in Sect. "Writing of Retrieved Information".

The `FunTuple` component also incorporates several essential counters to monitor the data processing. These counters include tracking the number of processed events, the count of non-empty events for each selected particle, and the tally of events with multiple candidates for each chosen particle. Upon completing the data processing, the results of these counters are displayed to the users. To ensure effective error handling, the component employs a custom error handling class that inherits from the `StatusCode` class implemented in `Gaudi`. This custom implementation enables the component to raise specific exceptions in targeted scenarios. For example, if a particular `ThOr` functor encounters difficulties and cannot compute an observable for a given event, the component raises an exception to promptly notify the user of the issue. Additionally, the `FunTuple` component takes measures to validate the input attributes both on the Python and c++ sides, ensuring the correctness of the provided data. Moreover, the development process includes the creation of several tests and examples, see Sect. "Test Suite, Examples and Performance".

Finding Decays in an Event

Given the distinct event models for reconstructed and simulated events, the `FunTuple` component employs two separate `Gaudi` tools for decay identification. Specifically, `FunTuple_Particles` relies on the `Gaudi` tool [36] `DecayFinder` [37], while `FunTuple_MCParticles` utilises the `MCDecayFinder` tool [38]. Both of these tools utilise the `boost` library [39, 40] to parse decay descriptors. The names of particles used in the decay descriptor, along with their associated properties, are stored in the LHCb conditions database (`CondDB`) [41], and are retrieved through the `ParticlePropertySvc` [42] service.

---

[4] Both run and event numbers are used to uniquely identify an event in the LHC experiments.

Listing 4 Example definition of the `fields` attribute

```
1  # define fields to select decays in an event
2  # key: alias of the particle used as a prefix to name the TBranch
3  # value: decay descriptor syntax select particles
4  fields = {
5      "Bplus": "[B+ -> (J/psi(1S) -> mu+ mu-) [K+]CC ]CC",
6      "Jpsi" : "[B+ -> ^(J/psi(1S) -> mu+ mu-) [K+]CC ]CC",
7      "kaons": "[B+ -> (J/psi(1S) -> mu+ mu-) ^[K+]CC ]CC",
8  }
```

To isolate a particular decay process within an event and select a particle within the decay chain, the user is required to provide a `fields` attribute to either the `FunTuple_Particles` or the `FunTuple_MCParticles` instance. The `fields` attribute takes the form of a string dictionary. Here, the `key` corresponds to the particle alias, serving as a prefix to label the `TBranch` in the resulting output file. On the other hand, the associated `value` denotes the decay descriptor employed to filter and select the particles participating in a distinct reconstructed or simulated decay process. A practical illustration of the `fields` attribute configuration is shown in Listing 4.

A correct syntax for the decay descriptor is crucial in the selection of the particles within a given decay process. A straightforward decay descriptor such as `"B+ -> J/psi(1 S) K+"` is employed to select all decays of a B^+ meson into a $J/\psi(1S)$ meson and a K^+ meson. For the inclusion of charge-conjugate decays, users can encapsulate the decay descriptor in square brackets and append the `CC` keyword, such as `"[B+ -> J/psi(1 S) K+]CC"`. This syntax covers both $B^+ \rightarrow J/\psi(1S)K^+$ and $B^- \rightarrow J/\psi(1S)K^-$ decays. Alternatively, the `[]CC` notation can also be used around an individual particle, e.g., `"B+ -> J/psi(1 S) [K+]CC"`, encompassing both $B^+ \rightarrow J/\psi(1S)K^+$ and $B^+ \rightarrow J/\psi(1S)K^-$ decays.[5] To target a specific particle within a decay, the caret symbol (^) is employed. For instance, `"B+ -> J/psi(1 S) ^K+"` selects the K^+ meson, while excluding the caret symbol selects the parent particle. In cases of identical particles in the final state, the `FunTuple` component ensures distinct C++ objects for each identical particle instance. For example, `"B+ -> ^pi+ pi- pi+"` and `"B+ -> pi+ pi- ^pi+"` would choose two distinct instances of a π^+. In the context of simulations, the `FunTuple_MCParticles` component utilises the LoKi decay finder [43]. This finder offers the flexibility to incorporate various arrow types within the decay descriptor syntax [43, 44]. Each arrow type allows users to selectively include simulated particles based on distinct criteria. For instance, the `=>` arrow type signifies the inclusion of an arbitrary number of additional photons stemming from final state radiation of charged particles when matching the decay.

Retrieve Event and Decay Information

To extract essential information related to either the event or individual particles within a decay chain, users are required to furnish the `variables` or `event_variables` attribute to `FunTuple`. The `variables` attribute functions as a python dictionary in which the `key` corresponds to the particle name previously defined in the `fields` attribute. The corresponding `value` is an instance of a `FunctorCollection`, which acts as a collection of `ThOr` functors, effectively resembling a dictionary itself, with the `key` representing the variable name and the `value` denoting a `ThOr` functor. Within the context of the `FunTuple` component, these `ThOr` functors are just-in-time (JIT) compiled and employed on the particle instance to retrieve the desired information. Notably, a key labelled `ALL` holds a special significance within the definition of the `variables`. Any `FunctorCollection` associated with the `ALL` key is applied to all particles specified in the `fields` attribute. In contrast, the `event_variables` attribute takes the form of an instance of `FunctorCollection`. The enclosed `ThOr` functors are designed to provide information at the event level. The specifics of how to define the `variables` and `event_variables` attributes are illustrated in Listing 5.

The `FunTuple` component utilises the flexibility inherent in `ThOr` functors to extract a diverse array of information from the event. These functors are adaptable enough to accept multiple reconstructed objects as input, enabling the computation of associated information. For instance, consider the functor designed to calculate the flight distance of a particle. To achieve this, the functor takes both the reconstructed primary vertices and the reconstructed particle as input arguments. The usage of this specific functor (`BPVFD`) is shown in Listing 5.

The functors support all fundamental mathematical operators, including addition, subtraction, multiplication, and division. In addition, they can undergo transformations such as `fmath.log(F.CHI2/F.NDOF)`, which, when applied to a reconstructed track, yields the track's

---

[5] The charge-violating decays are often reconstructed at LHCb to serve as proxies for the study of sources of background.

Listing 5 Example definition of the `variables` and `event_variables` attribute

```python
# import ThOr functor library
import Functors as F
# import the FunctorCollection library
import FunTuple.functorcollections as FC
# import function to get TES location of PVs
from PyConf.reading import get_pvs

# variables for "Bplus" defined in the "fields"
b_vars = FunctorCollection()
# store the flight distance of candidate B relative to the primary
    vertex that best aligns with the origin of candidate B.
pvs = get_pvs()
b_vars["BPVFD"] = F.BPVFD(pvs)

# variables for "Kaons" defined in the "fields"
kaon_vars = FunctorCollection()
kaon_vars["PT"] = F.PT

# variables for "ALL" particles defined in "fields"
all_vars = FunctorCollection()
all_vars["ETA"] = F.ETA

# define decay-level variables
variables = {
  "Bplus": b_vars,
  "Kaons": kaon_vars,
  "ALL": all_vars,
}

# define event-level variables,
# for example number of primary vertices
# and add FunctorCollection "SelectionInfo"
# that stores trigger configuration key (TCK) and
# decisions of "Hlt1LineName" trigger line
event_variables = FunctorCollection()
event_variables["nPVs"] = F.nPVs
evt_variables += FC.SelectionInfo(selection_type="Hlt2",
    trigger_lines=["Hlt1LineName"])
```

χ^2 per degree of freedom. Furthermore, the output from one `ThOr` functor can be passed as input to other `ThOr` functors through a mechanism known as *composition*. This proves particularly advantageous when users seek to compute an observable that relies on the outcomes of other observables. All these functionalities are harnessed to provide users with an range of observables via a pre-defined `FunctorCollection` instance, which is intended for use in conjunction with `FunTuple`. An illustrative example is the `SelectionInfo` collection, which gathers the functors employed to store the trigger configuration key (TCK) and the event's trigger line decision. Listing 6 outlines the definition of this collection, with its application showcased in Listing 5.

In this listing, the `SelectionInfo` collection is designed to take two main inputs: the type of selection, which can be any of the three stages (`Hlt1`, `Hlt2`, or `Sprucing`), and a list of trigger or Sprucing lines. In response, it generates a `FunctorCollection` that incorporates two functors: `F.TCK` for storing TCK information and `F.DECISION` for storing the trigger decision of the specified selection line. Such collections do not expose the users to the technical intricacies involved in retrieving the requested information. In this particular case, the involved functors require the `DecReport` object, which is obtained from the `DaVinci` framework via the `get_decreports` function. Furthermore, users maintain the flexibility to add, merge or remove observables within these collections, enabling them to create their customised collections. Multiple collections have been developed and continue to be actively expanded, accompanied by relevant unit tests within the `DaVinci` framework.

Listing 6 Example definition of `SelectionInfo` collection

```python
from GaudiConf.LbExec import HltSourceID
import Functors as F
from PyConf.reading import get_decreports

def SelectionInfo(*,
  selection_type: HltSourceID,
  trigger_lines: list[str]) -> FunctorCollection:
  """
  Event-level collection for tupling trigger/Sprucing information.

  Args:
      selection_type (HltSourceID): Name of the selection type i.e.
          "Hlt1" or "Hlt2" or "Spruce". Used as branch name prefix
          when tupling and as source ID to get decision reports.
      trigger_lines (list(str)): List of line names for which the
          decision is requested.

  Returns:
      FunctorCollection: Collection of functors to tuple
          trigger/Sprucing information.
  """

  # get selection type
  selection_type = HltSourceID(selection_type)

  # get decreports
  dec_report = get_decreports(selection_type)

  # check that the code ends with decision
  trigger_lines = [s + "Decision" if not s.endswith("Decision") else
      s for s in trigger_lines]

  # create trigger info dictionary
  trigger_info = FunctorCollection({
      selection_type.name + "_TCK": F.TCK(dec_report),
      l: F.DECISION(dec_report, l) for l in trigger_lines
  })
  return trigger_info
```

Writing of Retrieved Information

The `ThOr` functors, utilised for retrieving reconstructed or truth-level information, are capable of encapsulating data in a diverse range of formats. These functors can not only return basic c++types, but also yield complex objects pertaining to the LHCb software framework. Subsequently, the extracted information is recorded within the ROOT file, where each `TBranch` corresponds to a scalar or an array of basic c++types. `FunTuple` accommodates diverse data object types returned by `ThOr` functors. An illustrative example is the functor `F.STATE`, which retrieves the complete state of a reconstructed track, i.e. instance of `LHCb::State`, which includes information on track position, charge, momentum, track slopes, and the associated covariance matrix. `FunTuple` processes this returned class instance, enabling the writing of multiple observables into the ROOT file from a single functor. In this context, `FunTuple` supports various variable types, and the list is rapidly expanding. These include three vectors, four vectors, `SIMD` versions of arrays, matrices of both symmetric and non-symmetric nature with arbitrary dimensions, containers spanning arbitrary dimensions, various enumerations, e.g. vertex type, track state, as well as $std :: optional < T >$ constructs and $std :: map < std :: string, T >$ structures, where T represents any of the supported types. Additionally, extending support for other custom classes is remarkably straightforward.

As of the preparation of this document, the `FunTuple` component utilises the `GaudiTupleAlg` tool [42], which registers an entry in the ROOT file in a thread-safe manner. However, this tool does not provide full support for various complex data objects returned by `ThOr` functors; such support is exclusively offered by `FunTuple`. The transition to

ROOT's `RNTuple` is planned for the future with subsequent retirement of the `GaudiTupleAlg` tool.

Test Suite, Examples and Performance

`FunTuple` includes an extensive set of examples and tutorials for users, along with a dedicated test suite based on `pytest` [45]. Both unit tests and "physics tests" are crafted to assess various functionalities of the component, ensuring its reliability. Additionally, an application test accompanies each example job run in continuous integration, serving to guarantee correct functionality consistently.

Comprising just over 100 unit tests and some 40 "physics tests", the test suite currently in place evaluates various aspects of the FunTuple behaviour. These include checking for appropriate error messages in case of incorrect configurations, ensuring correct output with specified settings, validating expected numbers written to the ROOT file, testing the behaviour of `FunctorCollections`, assessing the output of FunTuple when run with different event models, and more. The test coverage for both `FunTuple` and the decay finder stands at an impressive 100%.

While a comprehensive performance analysis of `FunTuple` is not the focus of this paper, a brief overview is provided. In offline analysis, computing hundreds of observables is common. Recording 740 observables using ThOr functors for 1000 events takes 3 min, with JIT compilation of about 200 functors taking 84 s. Post-compilation, a functor cache is created, reducing overhead in both online and offline data processing. The Python front-end of `FunTuple` assists in early error detection in configurations, and the performance impact from combining C++/Python is minimal relative to functor execution time.

Interface with Other `Gaudi` Algorithms

In the LHCb framework, the execution of multiple algorithms within the offline data processing pipeline is a common necessity. Notable examples of such algorithms encompass the `DecayTreeFitter`[46], which fits complete decay chains with optional primary vertex constraints or mass constraints on intermediary states; the `MCTruthAndBkgCatAlg` algorithm [27], which is used to extract truth-level information from reconstructed objects in simulations; the `ParticleCombiner` algorithm [27], for combining basic particles into composite entities; among others. These algorithms can be employed in conjunction with the `FunTuple` component to process and store data. A practical illustration of `FunTuple` in synergy with `DecayTreeFitter` and `MCTruthAndBkgCat` is presented in Listing 7.

In this listing, the `DecayTreeFitter` and `MCTruthAndBkgCat` algorithms operate on reconstructed $B^+ \rightarrow J/\psi(1S)K^+$ decays. Under the hood, both algorithms construct a relation table linking the reconstructed object with a related object that holds pertinent information. For `MCTruthAndBkgCat`, the related object is the associated simulation object, harbouring truth-level information; conversely, for `DecayTreeFitter`, the related object corresponds to the output of the decay tree fitting process. To extract the relevant information, the reconstructed object is mapped to the related object, and the `ThOr` functor is applied to the related object. This entire process is executed within the `__call__` method of both the `MCTruthAndBkgCat` and `DecayTreeFitter` algorithms. For example, in Listing 7, calling `MCTRUTH(F.FOURMOMENTUM)` establishes a mapping between the reconstructed $B^+ \rightarrow J/\psi(1S)K^+$ decay and the corresponding simulation object. Subsequently, the `F.FOURMOMENTUM` functor is employed on the simulation object to retrieve the true four-momentum of the `B+` meson. A similar approach is followed for the `DTF(F.FOURMOMENTUM)`, with the distinction that the four-momentum of the B^+ meson is stored following the decay tree fit, incorporating mass constraint on the $J/\psi(1S)$ meson and primary vertex constraint.

The interaction between `FunTuple` and other `Gaudi` algorithms is fortified by a fail-safe mechanism. When either of the algorithms encounters failure, such as the absence of corresponding truth-level information or unsuccessful decay tree fitting, the `ThOr` functors and `FunTuple` are equipped to handle the situation. If the `ThOr` functor returns data of floating-point type, the `FunTuple` component automatically records Not a Number (`NaN`) in the ROOT file. Conversely, if the `ThOr` functor returns an integral type, the invalid value needs to be explicitly defined using the `F.VALUE_OR` functor, exemplified in Listing 7.

Summary and Conclusions

This paper introduces the `FunTuple` component, designed to support offline data processing for the LHCb experiment during the current Run 3 and subsequent runs. Its primary purpose is to facilitate the storage of experiment-acquired data in the `ROOT` format, optimising it for subsequent offline analysis. Currently, the component plays a vital role in various early measurement analyses of LHCb data collected during the current Run 3 data taking period. An example of the processed data using `FunTuple` is displayed in Fig. 3, showcasing the reconstructed mass of the $J/\psi(1S) \rightarrow \mu^-\mu^+$ decay from LHCb data gathered in 2022 during commissioning [47]. The figure shows the signal $J/\psi(1S) \rightarrow \mu^-\mu^+$ component in red-filled histogram and the background component in dotted purple line. The

Listing 7 Usage of the truth-matching (`MCTruthAndBkgCat`) and decay tree fitting (`DecayTreeFitter`) algorithms in conjuntion with `FunTuple`. Note that the `FunTuple` definition shown in Listing 1 does not change

```python
from DecayTreeFitter import DecayTreeFitter
from DaVinciMCTools import MCTruthAndBkgCat
import Functors as F
from PyConf.reading import get_pvs

# get the TES location of the input data with
# reconstructed "B+ -> J/psi(1S) K+" decays
input_data = get_particles(f"/Event/HLT2/BToJpsiK/Particles")

#get the reconstructed pvs
pvs = get_pvs()

# define an instance of MCTruthAndBkgCat algorithm for
    truth-matching.
# Arguments include:
# - name: User-specifed name
# - input_data: TES location of the input data
MCTRUTH = MCTruthAndBkgCat(name="MCTRUTH", input_data=input_data)

# define an instance of DecayTreeFitter for fitting the decay chain
# Arguments include:
# - name: User-specifed name
# - (optional) mass_constraint: Mass constraint on intermediate
    state (in this instance J/psi(1S))
# - (optional) input_pvs: TES location of reconstructed primary
    vertices to apply primary vertex constraint
# - input_data: TES location of the input data
DTF = DecayTreeFitter(name="DTF", mass_constraints=["J/psi(1S)"],
    input_pvs=pvs, input_data=input_data)

# define the B candidate variables to be passed to FunTuple
# Note: The "F.VALUE_OR" functor specifies an invalid value to be
    written to ROOT file in the case of no corresponding truth-level
    information. For functors returning floating point types such as
    components of F.FOURMOMENTUM, this is automatically chosen to be
    "NaN" by FunTuple
b_vars = FunctorCollection()
# add truth-level information
b_vars["TRUE_ID"] = F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID)
b_vars["TRUE_FOURMOM"] = MCTRUTH(F.FOURMOMENTUM)
# add decay tree fitter information
b_vars["DTF_FOURMOM"] = DTF(F.FOURMOMENTUM)
```

background component involves random combinations of muons from different part of the event. The total fit component, composed of both signal and background, is shown in solid blue line and the data points are shown in black dots. The number of signal candidates is estimated to be $N_{J/\psi(1S)} = 2354 \pm 93$ with mass $m_0 = 3093.6 \pm 0.2 \,\mathrm{MeV}/c^2$ and width $\sigma = 9.1 \pm 0.2 \,\mathrm{MeV}/c^2$ to be consistent with the known $J/\psi(1S)$ mass and width [48].

Furthermore, the `FunTuple` component is built upon the `Gaudi` functional framework, and it offers a user-friendly `Python` interface. Its templated design enables it to accommodate various types of input data, including reconstructed and simulated events, and it supports the processing of both event-level and decay-level information. Additionally, this templated design allows the component to support new event models, based on *SoA* data structure in the future, facilitating vectorised processing of data. Of particular importance is its ability to ensure equivalence between trigger-computed observables and those subjected to offline analysis. This achievement is made possible through the integration of the `ThOr` functors, adept at computing topological and kinematic observables. Users also have substantial flexibility, enabling them to personalise the range of observables stored within the `ROOT` file. The component is also thoroughly validated through a series of unit-tests and `pytest` tests to ensure its reliability. In conclusion, the unique attributes of the
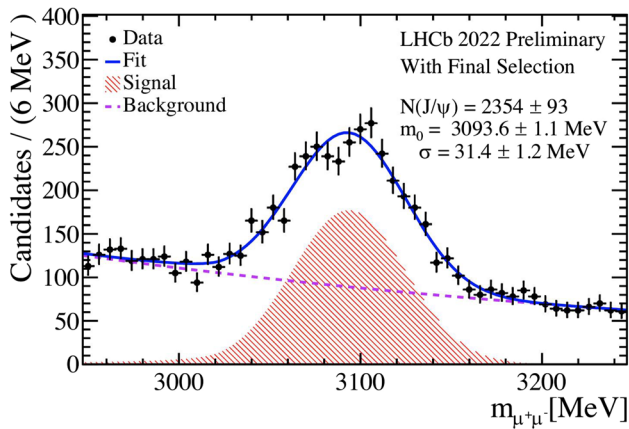
Fig. 3 Invariant mass of the $(\mu^- \mu^+)$ system showing the $J/\psi(1S)$ peak for LHCb data collected during the current Run 3 commissioning data taking period in 2022 [47]

`FunTuple` component establish it as a robust tool for offline data processing at the LHCb experiment making it essential for Run 3 and beyond.

Data availability The software component developed here is open source and is available in the gitlab via the link: https://gitlab.cern.ch/lhcb/DaVinci. The tests and examples use LHCb Run 3 data, which is not accessible openly.

References

1. LHC Machine (2008) JINST 3:S08001. https://doi.org/10.1088/1748-0221/3/08/S08001
2. LHCb Collaboration, Alves AA Jr et al (2008) The LHCb detector at the LHC. JINST 3: S08005. https://doi.org/10.1088/1748-0221/3/08/S08005
3. LHCb Collaboration, Aaij R et al (2015) LHCb detector performance. Int J Mod Phys A30: 1530022. https://doi.org/10.1142/S0217751X15300227arXiv:1412.6352
4. LHCb Collaboration, Aaij R et al. The LHCb Upgrade I, arXiv:2305.10515, to appear in JINST
5. Aaij R et al (2019) A comprehensive real-time analysis model at the LHCb experiment. JINST 14:P04006. https://doi.org/10.1088/1748-0221/14/04/P04006. arXiv:1903.01360
6. LHCb Collaboration, Aaij R et al (2022) Measurement of the $B^0_s \to \mu^+\mu^-$ decay properties and search for the $B^0 \to \mu^+\mu^-$ and $B^0_s \to \mu^+\mu^-\gamma$ decays. Phys Rev D105: 012010. https://doi.org/10.1103/PhysRevD.105.012010arXiv:2108.09283
7. LHCb Collaboration, Aaij R et al (2020) Measurement of CP-averaged observables in the $B^0 \to K^{*0}\mu^+\mu^-$ decay. Phys Rev Lett. 125: 011802. https://doi.org/10.1103/PhysRevLett.125.011802arXiv:2003.04831
8. LHCb Collaboration, Aaij R et al (2023) Test of lepton flavour universality using $B^0 \to D^{*-}\tau^+\nu_\tau$ decays, with hadronic τ channels. Phys Rev D108: 012018. https://doi.org/10.1103/PhysRevD.108.012018arXiv:2305.01463
9. LHCb Collaboration, Aaij R et al. Measurement of the ratio of branching fractions $R(D^*)$ and $R(D^0)$, arXiv:2302.02886, submitted to Phys Rev Lett
10. Skidmore N, Rodrigues E, Koppenburg P (2022) Run-3 offline data processing and analysis at LHCb. PoS EPS-HEP2021 792. https://doi.org/10.22323/1.398.0792
11. LHCb Collaboration (2018) Computing model of the upgrade LHCb experiment, CERN-LHCC-2018-014. http://cdsweb.cern.ch/search?p=CERN-LHCC-2018-014&f=reportnumber&action_search=Search&c=LHCb
12. Tsaregorodtsev A et al (2010) DIRAC3: the new generation of the LHCb grid software. J Phys Conf Ser 219:062029. https://doi.org/10.1088/1742-6596/219/6/062029
13. FunTuple GitLab Repository. https://gitlab.cern.ch/lhcb/Analysis/-/tree/v41r15/Phys/FunTuple. Analysis v41r15, Accessed 02 Nov 2022
14. Moore Project. https://gitlab.cern.ch/lhcb/Moore. Accessed 19 Aug 2023
15. Brunel Project. https://gitlab.cern.ch/lhcb/Brunel. Accessed 19 Aug 2023
16. DaVinci Project. https://gitlab.cern.ch/lhcb/DaVinci. Accessed 19 Aug 2023
17. Analysis project. https://gitlab.cern.ch/lhcb/Analysis/-/tree/v22r7?ref_type=tags. Accessed 19 Aug 2023
18. Belyaev I et al (2004) Python-based Physics Analysis Environment for LHCb. 2004. https://inspirehep.net/literature/928906
19. Bender Project. https://gitlab.cern.ch/lhcb/Analysis/-/tree/v22r7?ref_type=tags. Accessed 19 Aug 2023

20. LHCb Collaboration, Hennequin A, De Cian M, Esen S. Fast and flexible data structures for the LHCb Run 3 software trigger, https://doi.org/10.5281/zenodo.8119864arXiv:2307.03689

21. Aaij R et al (2020) Allen: a high level trigger on GPUs for LHCb. Comput Softw Big Sci 4:7. https://doi.org/10.1007/s41781-020-00039-7. arXiv:1912.09161

22. LHCb Collaboration, Aaij R et al. The LHCb upgrade I, arXiv:2305.10515

23. Fitzpatrick C, Gligorov VV (2014) Anatomy of an upgrade event in the upgrade era, and implications for the LHCb trigger. CERN, Geneva

24. Reiss F (2023) Real-time alignment procedure at the LHCb experiment for Run 3. http://cds.cern.ch/record/2846414

25. ThOr Functors.https://lhcbdoc.web.cern.ch/lhcbdoc/moore/master/selection/thor_functors.html. Accessed 02 Nov 2022.

26. Nolte N (2021) A selection framework for LHCb's upgrade trigger, 2020. https://cds.cern.ch/record/2765896. Presented 22 Feb

27. Rec Project. https://gitlab.cern.ch/lhcb/Rec. Accessed 19 Aug 2023

28. Lopez-Gomez J, Blomer J (2023) RNTuple performance: status and outlook. J Phys Conf Ser 2438:012118. https://doi.org/10.1088/1742-6596/2438/1/012118 (arXiv:2204.09043)

29. Brun R, Rademakers F (1997) ROOT: an object oriented data analysis framework. Nucl Instrum Meth A 389:81. https://doi.org/10.1016/S0168-9002(97)00048-X

30. LHCb Collaboration (2020) RTA and DPA dataflow diagrams for Run 1, Run 2, and the upgraded LHCb detector. https://cds.cern.ch/record/2730181

31. Analysis Productions project. https://lhcb-ap.docs.cern.ch/index.html. Accessed 19 Aug 2023

32. Standard C++, version C++17. https://isocpp.org/. Accessed 19 Aug 2023

33. Barrand G et al (2001) GAUDI—a software architecture and framework for building HEP data processing applications. Comput Phys Commun 140:45. https://doi.org/10.1016/S0010-4655(01)00254-5

34. Python Software Foundation. Python Language Reference, version 3.9, https://www.python.org/. [Online; accessed 19-Aug-2023]

35. LHCb Collaboration, Li P (2022) Real-time analysis in Run 3 with the LHCb experiment. PoS EPS-HEP2021 (2022) 829, https://doi.org/10.22323/1.398.0829

36. Clemencic M et al (2010) Recent developments in the lhcb software framework gaudi. J Phys Conf Ser 219:042006. https://doi.org/10.1088/1742-6596/219/4/042006

37. Barre S, Mathad, A. Decay finder algorithm for reconstructed particles in Run 3 at the LHCb experiment. https://cds.cern.ch/record/2837189. [CERN-STUDENTS-NOTE-2022-211]

38. LHCb Project. https://gitlab.cern.ch/lhcb/LHCb. Accessed 19 Aug 2023

39. Boost.Regex 7.0.1. https://www.boost.org/doc/libs/1_80_0/libs/regex/doc/html/index.html. Accessed 19 Aug 2023

40. Joel de Guzman, HK (2011) Qi—Writing Parsers. https://www.boost.org/doc/libs/1_80_0/libs/spirit/doc/html/spirit/qi.html. Accessed 16 Sept 2022

41. LHCb Conditions Database. https://gitlab.cern.ch/lhcb-conddb. Accessed 19 Aug 2023

42. Gaudi framework. https://gitlab.cern.ch/lhcb/Gaudi. Accessed 19 Aug 2023

43. LoKi framework. https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders. Accessed 19 Aug 2023

44. LHCb collaboration (2022) Grammar in short: Arrows. https://twiki.cern.ch/twiki/bin/view/LHCb/FAQ/LoKiNewDecayFinders#Arrows. Accessed 16 Sept 2022

45. Krekel H et al (2004) pytest. https://docs.pytest.org/en/7.1.x/

46. Hulsbergen WD (2005) Decay chain fitting with a Kalman filter. Nucl Instrum Meth A552:566. https://doi.org/10.1016/j.nima.2005.06.078. arXiv:physics/0503191

47. LHCb Collaboration (2022) Jpsi2MuMu 2022 mass figure. https://cds.cern.ch/record/2867664

48. Particle Data Group, P. A. Zyla et al (2020) Review of particle physics. Prog Theor Exp Phys. 2020: 083C01. https://doi.org/10.1093/ptep/ptaa104http://pdg.lbl.gov/