24 August 2023

CERN LHCb RUNDB Operation Plots Project

Struan Bowie

Summary

This is a report on the work accomplished over a 12 week period (05/06/23 - 25/08/23) by Struan Bowie, a member of the CERN 2023 Summer Student Program. The aim of this project was to produce automatically generating, interactive operation plots that track LHCb beam performance over a variety of parameters and time ranges. This has been integrated into the pre-existing RUNDB web page [1], where user inputs can customise the data type, plot type and scale the dimensions of the plot quickly and intuitively. Now, data from each year, fill and run can be inspected and assessed for suitability as soon as it is uploaded to the Oracle database. This greatly improves efficiency over the previous workflow which involved many cumbersome and time consuming steps, including: manually exporting the data, plotting locally, saving each plot as a PNG and finally uploading the graphs to the web page as static images [2].

Frameworks and Libraries

The project involved both front and back-end development. Python's Django was chosen as the back-end framework [3], allowing for quick and scalable development, while the D3.js library was used to plot and render the data [4]. This library was selected for its reactivity and ability to rapidly process large data sets.

Django is a popular web development framework. Not only is it python based (one of the most widely used languages), but it comes with a wide range of built-in tools and features that greatly increase the speed and productivity of the workflow. For this project, the Object-Relational Mapping (ORM) capabilities of Django is where it proved most useful. This provides the ability to easily interact with a database, without the need for complex SQL queries. In addition to this it has many security features such as cross-site request forgery (CSRF) that protects against common security weaknesses.

D3 stands for data driven documents and differs from more conventional plotting libraries such as Chart.js or Plotly. It allows for much more customisation and experimentation when visualising data compared to the standard scatter plot or bar chart options given in other libraries. This is because D3 renders its plots as scalable vector graphics (SVG's) rather than the HTML5 canvas used by Chart. Inputted data can then be used to alter and an the position, shape, size , colour and many more attributes of each SVG. While the use of D3 was no strictly necessary for this project, it provided a valuable opportunity to expand skills in data visualisation outside of the generic plotting libraries.

Data Flow and Processing

The data is stored in, and retrieved from, two Oracle data bases (DB's): rundb_int12 (rundb) and wincc_lb_lhbon (wincc) [5]. Before accessing the website, the user must be signed in to an active CERN account in order to be recognised as a valid reader. Fig.1 and Fig.2 show the data acquisition path. They are constructed as class method attributes that were added to the Django Models for the relevant tables and views. These class methods are themselves, a series of queries to the DB.

It can be seen that not all the required information can be extracted in a single query. For example, in the code shown below and Fig.2 the valid ELEMENT_ID for each data point is first found in the ELEMENTS_ALL table. It is then paired with its name label. The time range of the fill is calculated and used as a filter in the Eventhistory query along with the ID's. Without this, the query spans the entire data base rather than just the values for a specific run or fill. Finally, the ID's are compared and replaced with the corresponding name label. This is an important step as the name is required when distinguishing different data points from one another in the rendering stage of the process.

```
// models.py

class Rundbruns(models.Model):
    ...
```

```
@classmethod
def get_run_plot_data(cls, fillid, element_matches):
    id_values = ElementsAll.objects.using('wincc')
                    .filter(element_name__in=element_matches)
                    .filter(valid_till=None)

    extracted_ids = extract_ids(id_values)
    tups = tuples(element_matches, extracted_ids)
    fill_all = Rundbfills.objects.get(fill_id__exact=fillid)
    time_range = [fill_all.timestamp -
                  timedelta(seconds=int(fill_all.time_total)),
                  fill_all.timestamp]

    data_raw = Eventhistory.objects.using('wincc')
                .values('element_id', 'ts', 'value_number')
                .filter(element_id__in=extracted_ids)
                .filter(ts__range=(time_range[0], time_range[1]))

    data = pair_data(data_raw)
    return data
```

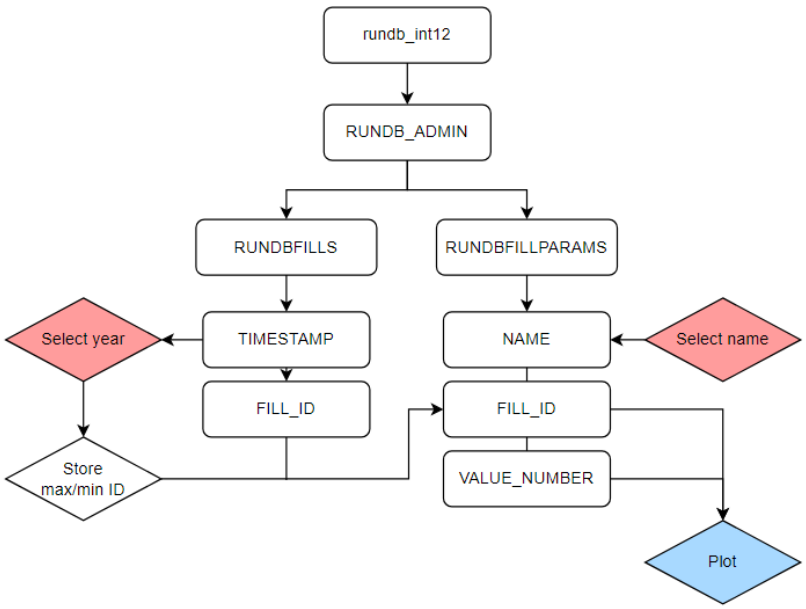

Figure 1: This flowchart shows the data acquisition path through the rundb. While user inputs control the final query, the same path is used each time to pair data points and package together.

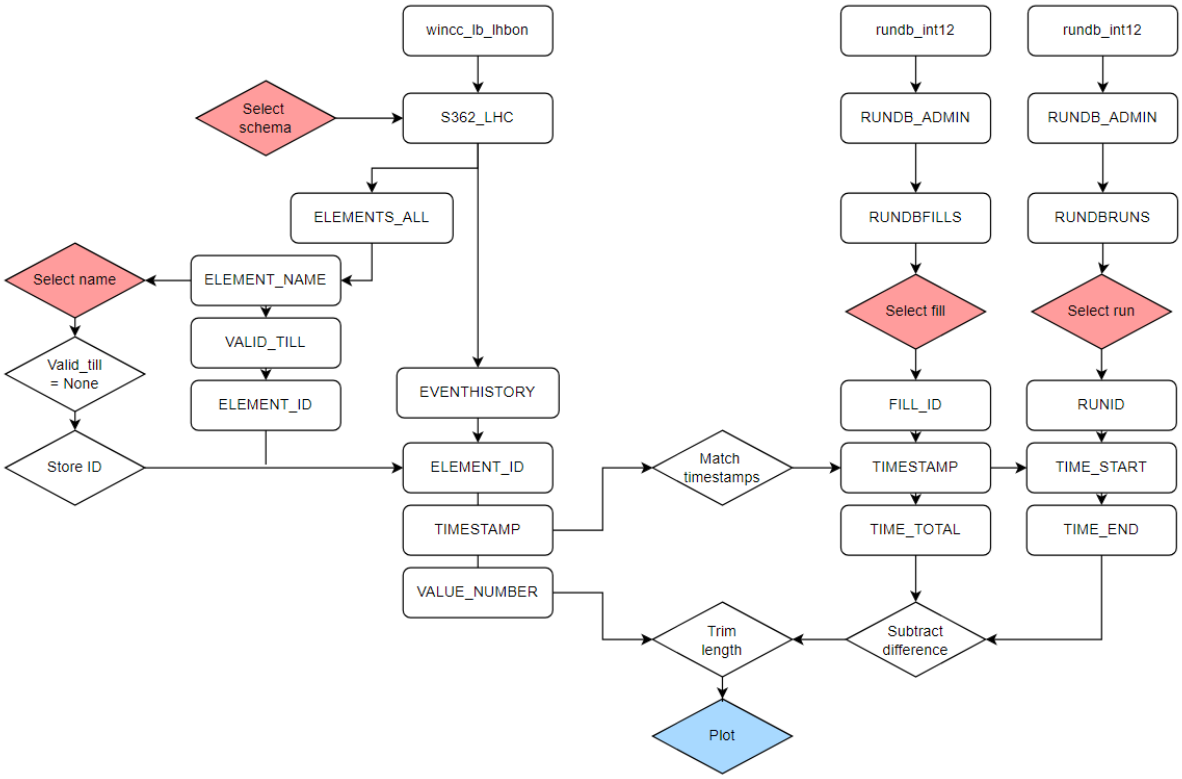

Figure 2: This is a flowchart of the wincc related queries. It has more steps than rundb which are explored in more detail in the report. It can be seen that the wincc queries still depend on the rundb to acquire time ranges for both the fills and runs.

The red boxes indicate variables that must be selected by the user, changing the parameters of the queries. For first time page loads the default options are set as the most recent additions to the DB. After processing the inputs and queries the data is packaged as an array of objects in the form:

```
data = [{'name':        peakLumi,
         'timestamp':   2023, 8, 21, 17, 6
         'value':       639.45},
        {...},
        {...},
         ...]
```

The blue boxes show the final data being extracted as the form described above. The class method is then called in the Django views.py file, stored in the templates context dictionary, passed to the template HTML before finally being converted to the JSON format and loaded into the external JavaScript plotting files.

```
// views.py
```

```
def export(request):
    fill_list = rundb_fill_list_query()
    ...

    if request.method == 'POST':
        selected_fill = request.POST.get('fill_selection')
        ...
    else:
        selected_fill = fill_list[0]
        ...

    elements_match = wincc_element_match(selected_elements)
    plot_data = Rundbruns.get_run_plot_data(selected_fill, elements_match)

    return render(request, 'rundb/rundb_export.html',
        {'form':export_form(request),
        'fill_list': fill_list,
        'selected_fill': selected_fill,
        ...
        'runs_plot_data': plot_data,
        })

// rundb_export.html

{% load static %}
<script type="module" src="{% static 'js/render.js' %}" defer></script>

<div class="span-23 box last">
    {{ runs_plot_data|json_script:"runs_plot_data_ID" }}
    <div id="plot_div"></div>
    ...
</div>
```

Plotting and Rendering

The plotting and rendering of the graph is separated into two files. Plot.js and Render.js.

In the former, a plot function is constructed. This is where D3 is used to process the inputted data in order to produce standard aspects of a graphs such as; legends, grid lines, axis ticks and titles. It is also where features like the zoom brush are held.

The render file handles all of the user inputted data. It is retrieved using JavaScript's getElementByID and linking to the appropriate drop down selection in the HTML form. Here is also where the data is loaded as a JSON from the HTML template and parsed

back into a usable format. JSON converts all values to strings so the data point values and timestamps need to be parsed back to floats and dates respectively.

Getter/Setter methods are chained on the plot function. They allow attributes of the graph (dot radius, axes titles, colours and other parameters) to be controlled externally to the function.

```
// plot.js

export const plot = () => {
    let data;
    ...

    const my = (selection) => {
        const x = d3
            .scaleLinear()
            .domain(d3.extent(data, xValue))
            ...
        ...
    }
}

my.data = function (_) {
      return arguments.length ? ((data = _), my) : data;
};

// render.js

import { plot } from "./plot.js";

const runs_plot_json = JSON.parse(document
                           .getElementById("runs_plot_data_ID")
                           .textContent);

const create_graph = (data, ..., type) => {
  plot_g
    .append("div")
    .append("svg")
    ...
    .call(
      plot()
        ...
        .data(data)
    )
}
```

As mentioned previously in the report, the graphs were constructed using the D3 library. Several features have been implemented to provide a fast and convenient experience when interpreting the data.

1. Zoom-able axes
2. Multi-variable plotting
3. Data point labels on hover
4. Choice of plot types
5. Quick and responsive variable selection
6. Dynamic legend

The zoom feature can be used by clicking and dragging the left mouse button over the desired section of the graph [Fig.3] [6]. After releasing, the axes will resize to fit the rectangle created by the user. To return to the default scale, the user must double click the left mouse button.

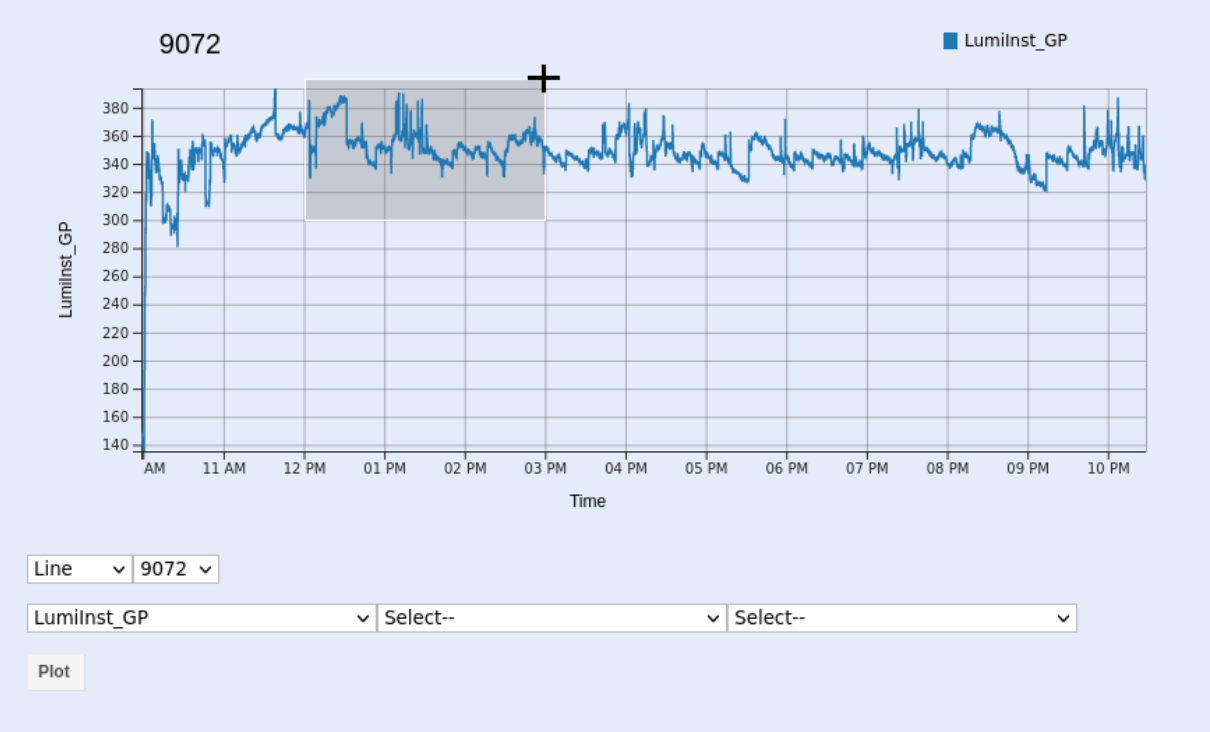

Figure 3: This figure demonstrates the zoom feature. Clicking and dragging the left mouse button produces a rectangular brush that can be used to re-scale the axes of the plot. Double clicking returns the plot to its default setting.

As can be seen in Fig.4 and Fig.5, multiple variables (up to three) can be selected and plotted on a single graph. This allows for easy comparisons between related variables such as the x, y and z components of the beam crossing angle.

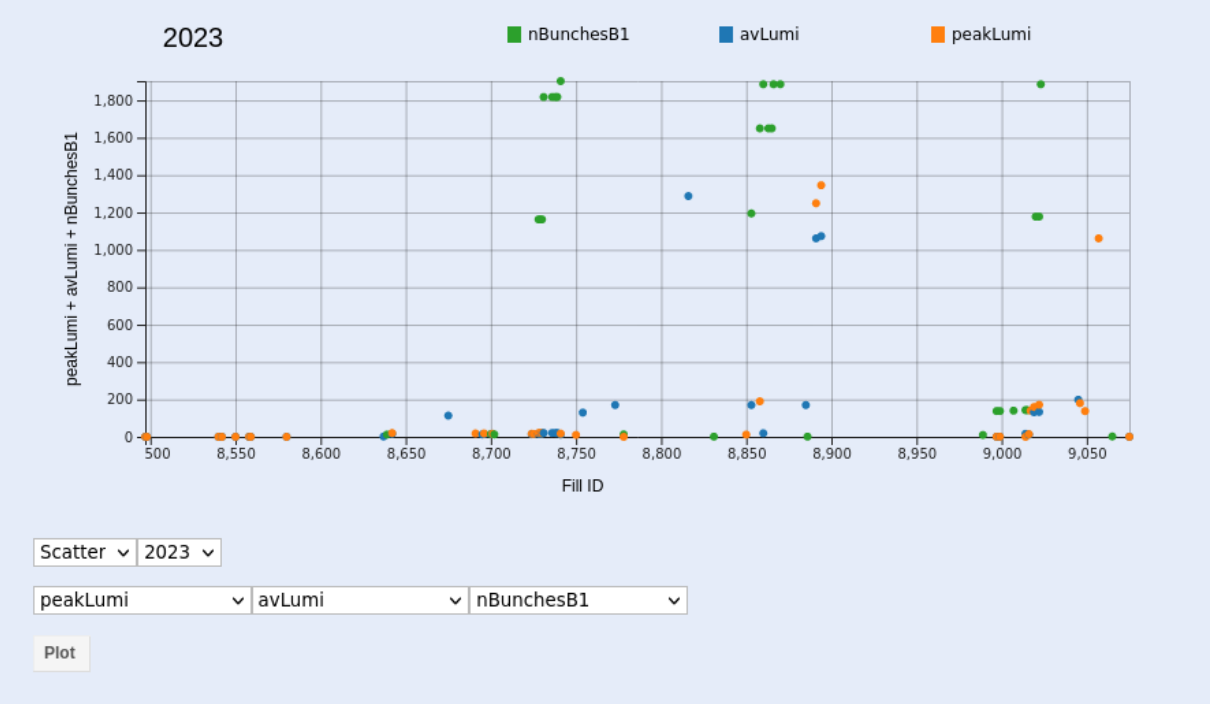

Figure 4: This plot has the maximum of three variables (nBunchesB1, avLumi and peakLumi) simultaneously plotted. These can be swapped and selected at will using the dropdown selections located below the graph. A legend at the top of the plot displays the corresponding colour of each variable.

Hovering over a data point produces a small label. This removes the issue of incorrectly reading off of the axis and can return the value of the specific data point up to the accuracy it is stored in the DB with. This compliments the colours as a method of distinguishing between variables as it also displays the name of the variable it belongs to.

There is also a choice between a line or a scatter graph. Depending on the users requirements, the plot type can be instantly swapped. This allows for higher levels of control, allowing for the plot to be tuned to the users needs and improving the overall experience.

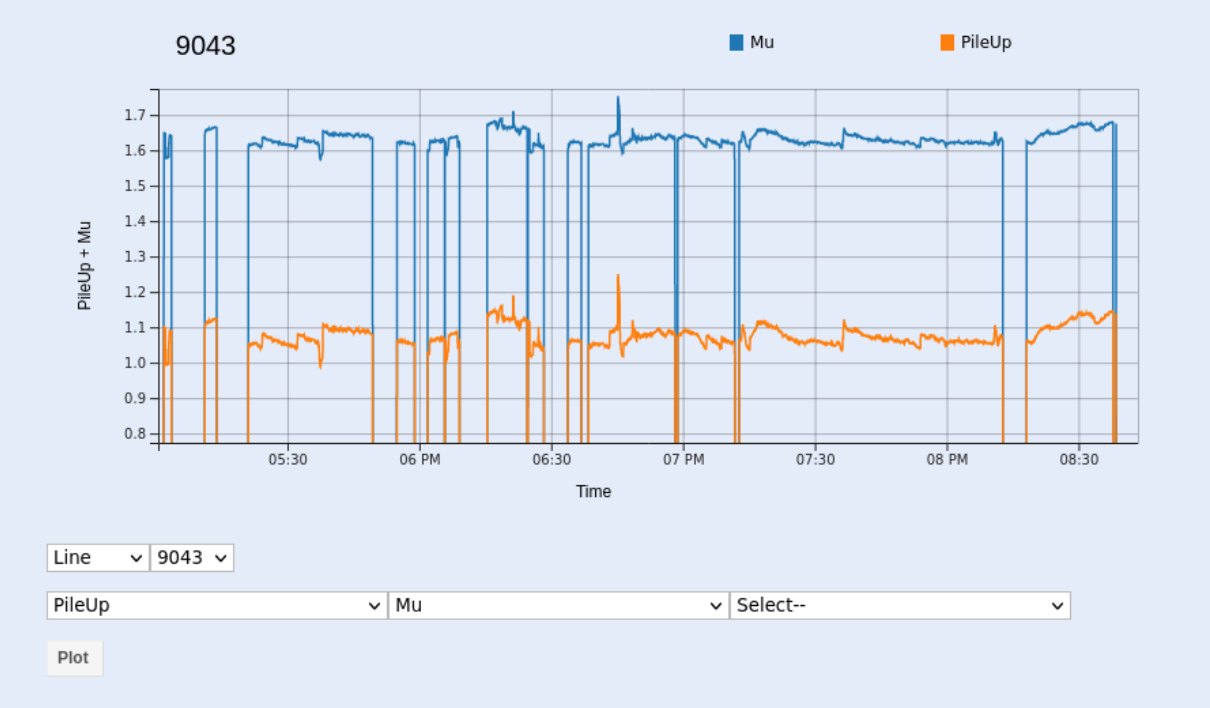

Figure 5: Here, the correlation between PileUp and Mu can be easily seen by selecting line graph.

Future Improvements

While the project has concluded after 12 weeks, there are some small amendments that could be made in the future to improve the overall user experience.

Firstly, generating a graph by clicking the plot button has an unintended effect on the preexisting table form. This form has several check boxes set by default, however, hitting plot refreshes the page and removes these selections. This causes the table (located below the plot) to be unconstrained by a time range, and lists all runs/fills listed on the DB. While this does not render the site unusable, it would be a generous quality of life update to fix this issue.

Secondly, the current zoom feature is, by default, a rectangle in both the X and Y axis. An improvement could be to limit the brush to only the X direction unless the user moves their mouse outside of some horizontal bars. At this point the brush would return to its current style, unconstrained in either axis. This is useful as, primarily, when zooming the user is concerned with only the temporal (X) axis.

Finally, when displaying multiple variables on a single plot it can be difficult to read the data if the scales vary too greatly. To address this it is possible to have multiple Y axes (one for each variable), that is scaled appropriately.

References

[1] CERN LHCb Collaboration. Run database. https://lbrundb.cern.ch/, Accessed on 24/08/2023.

[2] CERN LHCb Collaboration. Rundb operations plots. https://lbgroups.cern.ch/online/OperationsPlots/index.htm, Accessed on 24/08/2023.

[3] Django Software Foundation. Django documentation. https://docs.djangoproject.com/en/4.2/, Accessed on 24/08/2023.

[4] Observable. D3.js documentation. https://d3js.org/, Accessed on 24/08/2023.

[5] Oracle. Oracle database documentation. https://docs.oracle.com/en/database/, Accessed on 24/08/2023.

[6] D3.js Graph Library. Zooming in d3.js. https://d3-graph-gallery.com/graph/interactivity_zoom.html, Accessed on 24/08/2023.