Parkview

Performance Dashboard for Continuous Benchmarking of HPC Libraries

Chingun Ariunbat, Jamil Bagga, Walter Alexander Böttcher, Darius Schefer, Maximilian Schik

2021-07-21

Contents

1 Introduction

This document details the reworked architecture of *Parkview* and documents changes made to the previous design. Since most changes came in the shape of simplifying the architecture the end product itself is simpler and therefore much easier to maintain and extend, while still keeping all of the promised functionality. The most severe change is the handling of GitHub history data, since it no longer gets stored but instead fetched from the GitHub Api. Also almost all extension now happens on the backend, so for most extensions like new plot transformations no changes to the frontend are needed.

2 System Overview

The basic shape of the application stayed the same, with the backend handling data storage and processing, and the frontend requesting and visualizing data from the backend. Figure 1 gives a high level overview of all the important components.

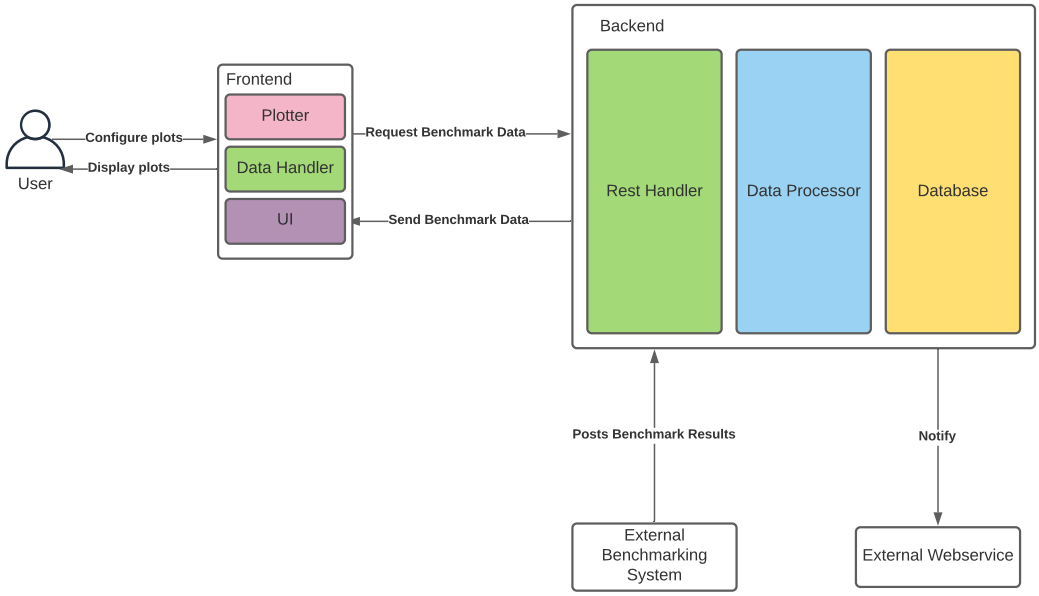

Figure 1: System Model

2.1 Backend

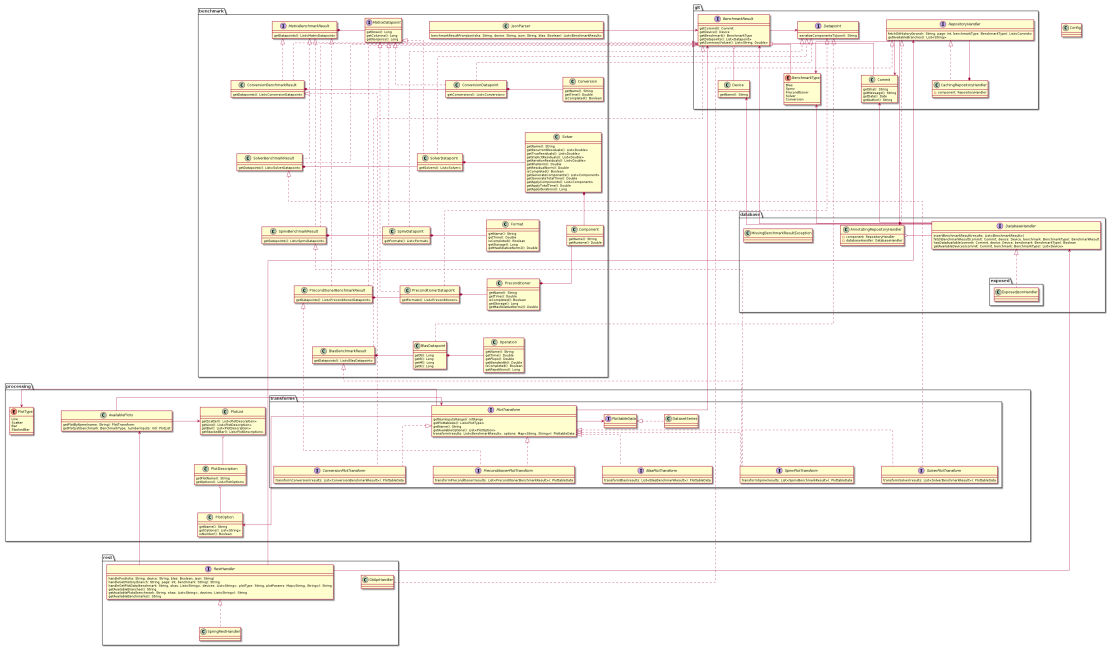

Figure 2: UML class diagram for the backend

The most important class/interface here is `PlotTransform`. It models transforms from benchmark results to plottable data. Every plot transformation implements this interface. Following is a more in depth description of the interface members:

- `val numInputsRange:IntRange`: Range of allowed number of inputs. Since some plots can only take a single benchmark result and some can take multiple, this has to be defined. It should be mentioned that Kotlin offers some syntactic sugar for ranges, for example the range from one to ten can be represented by just `1..10`.

- `val plottableAs:List<PlotType>`: Lists all plot types that can represent this transform. It uses the `PlotType` enum for that. This information is mainly for the frontend, since it has to know what plot to use.

- `val name:String`: Name of transform, it has to be unique across all registered plot transforms

- `fun getAvailableOptions(results:List<BenchmarkResult>):List<PlotOption>`: Returns a list of all available plot options, each containing a list of allowed values.

- `fun transform(results:List<BenchmarkResult>):PlottableData`: Function that does the actual transforming. Usually you don't override this exact function because you implement one of the "subinterfaces" like `SpmvPlotTransform` which already overwrote that function and does some type and error checking. Instead you would override `fun transformSpmv(results:List<SpmvBenchmarkResult>):PlottableData`

A quick example would be this snippet taken from `SpmvSingleScatterPlot`:

```
1  class SpmvSingleScatterPlot : SpmvPlotTransform {
2      override val numInputsRange = 1..1
3      override val plottableAs = listOf(PlotType.Scatter)
4      override val name = "spmvSingleScatterPlot"
5      override fun getAvailableOptions(results: List<BenchmarkResult>):
           List<PlotOption> = listOf(
6          PlotOption(
7              name = "yAxis",
8              options = listOf("bandwidth", "time"),
9          ),
10         PlotOption(
11             name = "xAxis",
12             options = listOf("nonzeros", "rows", "columns"),
13         ),
14     )
15
16     override fun transformSpmv(
17         benchmarkResults: List<SpmvBenchmarkResult>,
18         options: Map<String, String>
19     ): PlottableData {
20       // transform code
21     }
22 }
```

This is a transform fot the SPMV benchmark, so it implements the `SpmvPlotTransform` interface (inheritance/implementation in kotlin is symbolized by a colon). Since it takes only one benchmark result as an input we set `numInputsRange` to a range from one to one, so only one (line 2). This plot should only be plottable as a scatter plot, so in line 3 we set `plottableAs` to a list containing only the enum for scatter plots. We set the name in line 4 and in line 5 we set the available options. We create one `PlotOption` object per option, so one for the yAxis and one for the xAxis, each with its corresponding possible values. Those value later get represented with a dropdown menu by the frontend.

To register your (new) plot transforms, you have to add it to the corresponding list in `AvailablePlots`. You have to edit the actual code for that and recompile afterwards.

2.2 Frontend

2.3 Database

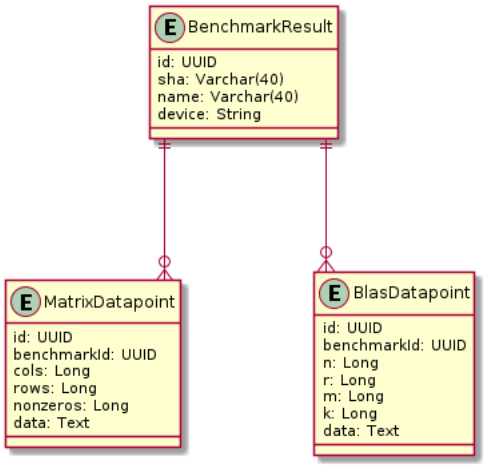

Figure 3: Entity Relation diagram

The previous database scheme consisted of many tables for each component of a benchmark result. This made the database accesses slow and lead to a very big memory footprint. The new database scheme uses only three tables and is much faster and compacter as a result of that. This was accomplished by storing the data for solvers, formats, operations etc. in JSON format. This only has the downside of adding complexity to the database handler when updating results with new datapoints.

3 API

/post

Name	Type	In	Description
sha	String	query	Sha for benchmark result
device	String	query	Name for benchmark result
blas	Boolean	query	Has to be true when uploading blas benchmark results
json	JSON	body	Contains the benchmark data