Demonstration Paper

Do Research, not Data Visualization! How to Create More Consistent Plots for Experimental Research Papers in Less Time

Justus Henneberg
Johannes Gutenberg University
Mainz, Germany
henneberg@uni-mainz.de

Felix Schuhknecht
Johannes Gutenberg University
Mainz, Germany
schuhknecht@uni-mainz.de

ABSTRACT

In the database community, research is accompanied by extensive experimental evaluation. This evaluation produces large amounts of result data which is typically visualized in form of various plots. Unfortunately, the path from the data to the corresponding plots is currently long, cumbersome, and frustrating. In our experience, this largely stems from the challenges that arise in keeping *all plots of a paper consistent with each other*: Ensuring that all properties of all plots, such as labeling, coloring, scales, or order of the presented results actually match each other requires a constant revisiting and readjusting of the redundant portions of a large number of plotting scripts. This unnecessarily eats away time from doing meaningful research. To address this problem, we present our Python framework *ChartGallery*, which we already successfully used in our group for previous research papers. It sets itself apart by not being yet another plotting library, but a framework that focuses on managing all plotting setups and styles of a paper in an organized way in order to compose various different plots with minimal effort and code redundancy.

1 INTRODUCTION AND MOTIVATION

Creating plots for a research paper is an overhead task and therefore should take little effort. At the same time, plots should be consistent, informative, and beautiful to make results more digestible for the reader. Unfortunately, current workflows fail to fulfill both requirements.

1.1 The Plotting Pipeline

To understand the problem, let us first outline the four steps that are carried out in one way or the other in any plotting pipeline:
(1) *Loading and preprocessing of the experimental results.* This step happens once per result set. The user must define the location of the result set (e.g. a directory path) and how to handle the input format (e.g. unformatted text, CSV, or a database). After loading, the dataset typically must be somehow preprocessed, such as setting or changing column names, or to average data across multiple runs.
(2) *Declaring the plot.* This step happens individually for each and every plot that will end up in the paper. It typically involves some further preprocessing of the result set, e.g. filtering out columns and rows that are not needed. Then, the type of plot must be set (e.g. a bar plot) and based on the chosen type, portions of the data are mapped to the components of the plot (e.g. to the individual groups, bars, and bar-stacks). Also, the name of the resulting plot is set here.
(3) *Formatting the plot.* This step typically does not happen for each and every plot individually, but at the granularity of groups of plots that will end up together in the paper (e.g. as multiple sub-figures of a figure), as they require a consistent visual style. This consists of setting various properties, such as axis labels, colors, hatching, and sizes. Further, this step can involve arranging legends, or setting the value ranges of the axes.
(4) *Exporting the plot.* This final step is typically identical for all plots, as every plot of the paper usually has to be in the same format. This might involve setting specific export options (e.g. PDF/A compliance), file extensions, and a common output destination (e.g. a path to an output folder).

1.2 The Typical Plotting Workflow

As we can see, the outlined four steps of each plotting pipeline are configured at very different granularities — while step (1) requires setup only for every distinct result set, step (2) varies from plot to plot, step (3) happens for each group of plots, and step (4) is set up globally.

Unfortunately, these different granularities are not properly reflected in the typical plotting workflow. The reason for this is that the workflow of research papers is highly demand-driven. When the first plot is required during research activity, the resulting plot script will likely contain all four steps. When the second similar plot is required, parts of the script are simply copied and adjusted, as this can be done quickly[1]. This continues until such a large number of plotting scripts exist that they start to deviate from each other in terms of consistency. What was fast and easy in the beginning eventually turns into a management nightmare: If something changes in the result set or plot formatting (e.g., different column names or axis labels), the change must be reflected in various places across various scripts. Periodically, scripts must be aligned to fight the various occurring inconsistencies[2].

2 COMPOSING WITH CHARTGALLERY

Instead, we advocate to streamline the plotting workflow with ChartGallery, our chart composing framework. We specifically designed ChartGallery to reflect the different configuration granularities of the plotting pipeline in a convenient and easy-to-use manner. On a high level, we allow the user to wrap the plot setup into a single object which can be applied globally or with individual plots. Plot setups can be freely composed and/or extended, which permits describing a large number of related plots with minimal code redundancy, and hence, high consistency.

2.1 Architecture

Figure 1 visualizes the core abstractions we chose for ChartGallery. Note the parallels to Section 1.1: The *core plotting function* only handles data pre-filtering and delegates to an appropriate library for plotting [2–5]. Since there is no direct interaction between ChartGallery and the plotting library, there are no compatibility restrictions. Data loading is also defined more abstractly for the same reason: A *data source* can be any Python callable that returns a data collection, usually a DataFrame variant. When the user runs the core plotting function, ChartGallery invokes

[1] which makes the PhD advisor happy as results are produced
[2] which make the PhD advisor very unhappy as they slow down research

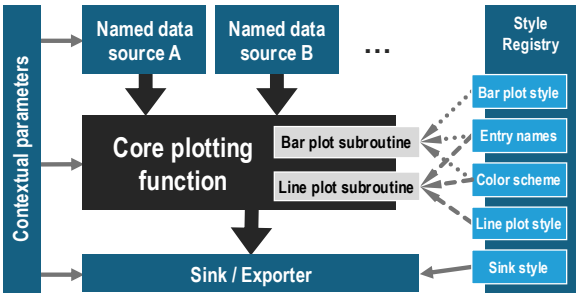


Figure 1: Architectural overview of ChartGallery.

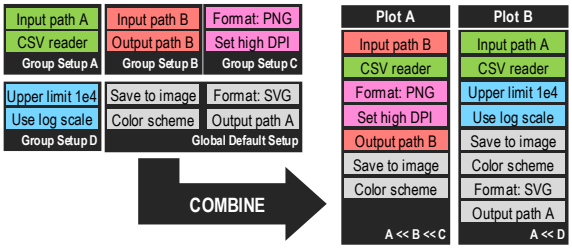


Figure 2: Combining setup objects in two different ways.

all data sources that are declared as parameters, and passes the resulting datasets as arguments. A common use would be reading a CSV file. The return value of the core plotting function is automatically handed over to the *sink*. Usually, this return value is the finished plot, and the sink exports it to disk. However, we will later show that sinks are more flexible than this. Any component can read from a set of string-to-value mappings, called *contextual parameters* and *styles*. Contextual parameters store arbitrary mappings, while styles define a fixed key set and a default value assignment. In practice, styles are designed to store documented parameters for fixed components, such as a plotting subroutine, or a complex sink, while parameters are intended for ad-hoc use.

Most plotting scripts will naturally converge toward these abstractions when moving shared code into helper functions or global constants. The key difference in our work is the fact that the entire context around the core plotting function (sources, sinks, parameters, shown in teal in Figure 1) is encapsulated as a single object, the *setup object*. Setup objects have three core properties: (1) They are reusable, (2) they are composable and (3) they can choose to define only parts of the context. Therefore, all properties that are typically shared between a subset of plots can be expressed as a setup object, and a selection of those setup objects can be composed to form the final setup that is associated with the function. Consequently, code duplication is reduced to almost zero, since it only happens when we assign setups to Python functions, and consistency between plots happens as a natural side-effect of re-using setups. As setup objects are plain Python objects, they can be stored in collections, serialized to disk, or assembled and returned by helper functions, if required. Figure 2 shows a high-level example where four setups are mixed and matched to form the contexts for two different plots. ChartGallery expresses composition as `a << b`, where duplicate settings are resolved by preferring those set in `b`. Defaults are applied whenever an option is never explicitly overridden.

3 DEMONSTRATION

In the following, we demonstrate how to create a series of consistent plots using ChartGallery using a concrete example[3]. We show the process step-by-step to mimic the typical workflow when creating plots for a research paper, where plots are designed incrementally and build upon each other.

3.1 Starting Situation

We use two datasets which were produced from experimental runs for our latest research paper [1], in which we proposed a new index structure for GPUs. Both datasets are materialized as CSV files (`rtx-pq.csv` and `rtx-group-size-scaling.csv`), where the individual columns consist of the experimental configuration and the associated measurements. While knowing the detailed schema is not required for following the example, the `rtx-pq.csv` dataset essentially captures a comparison of multiple index structures under different configurations (key width, percentage of hits, and so on), while the `rtx-group-size-scaling.csv` dataset measures the impact of various hyperparameters on our index structure alone (group size, structure optimizations, and so on).

Imagine we are now interested in visualizing the probe time contained in both datasets. The `rtx-pq.csv` dataset contains results for different index structures and is further divided into results for 32-bit keys and 64-bit keys (parameter `key_bits`), and we want to create two corresponding plots showing the results for all indexes while varying another parameter on the x-axis. The `rtx-group-size-scaling.csv` dataset contains results for both a scaled and an unscaled run (parameter `z_scale_log`), and we want to create two corresponding plots again. This time, each plot should vary two parameters on the x-axis, while showing the results for four different group sizes and optimization levels.

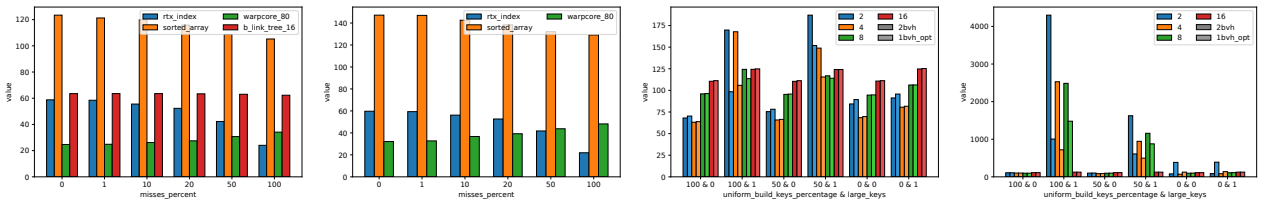
3.2 The Typical Approach

The straightforward way of creating these four plots (Figure 3) would likely look like Listing 1: For each plot, there is a separate section which reads the input file into a DataFrame, performs the necessary preprocessing, followed by the actual plotting, and dumping the plot. While this gets the job done, it obviously comes with severe code duplication due to the similarity of the plots: Apart from different output file names, each plot pair only differs in the filtering of the parameter `key_bits` or `z_scale_log`.

```
1  def read_data(filename):
2    data = pl.read_csv(filename)
3    return average_runs(data, "run", "VALUE")
4
5  data_c = read_data("input-data/rtx-pq.csv")
6  data = data_c.filter(pl.col("key_bits") == 32)
7  data = data.select("misses_percent", "index_type", "VALUE")
8  fig, ax = new_figure()
9  bar.bar_plot(data, ax, color_by="index_type")
10 fig.savefig("output-plots/compare_execution_time_32.pdf")
11 data = data_c.filter(pl.col("key_bits") == 64)
12 data = data.select("misses_percent", "index_type", "VALUE")
13 fig, ax = new_figure()
14 bar.bar_plot(data, ax, color_by="index_type")
15 fig.savefig("output-plots/compare_execution_time_64.pdf")
16
17 data_gs = read_data("input-data/rtx-group-size-scaling.csv")
18 data = data_gs.filter(pl.col('z_scale_log') == 0)
19 data = data.filter(pl.col('group_size_log').is_in([2, 4, 8, 16]))
20 data = data.select("uniform_build_keys_percentage", "large_keys",
21                    "group_size_log", "indexing_method", "VALUE")
22 fig, ax = new_figure()
23 bar.bar_plot(data, ax, color_by="group_size_log", shade_by="indexing_method")
24 fig.savefig("output-plots/group_size_time_unscaled.pdf")
25 data = data_gs.filter(pl.col('z_scale_log') == 25)
26 data = data.filter(pl.col('group_size_log').is_in([2, 4, 8, 16]))
27 data = data.select("uniform_build_keys_percentage", "large_keys",
28                    "group_size_log", "indexing_method", "VALUE")
29 fig, ax = new_figure()
30 bar.bar_plot(data, ax, color_by="group_size_log", shade_by="indexing_method")
31 fig.savefig("output-plots/group_size_time_scaled.pdf")
```

Code Listing 1: Default approach with code redundancy.

[3]Code available: https://infosys.informatik.uni-mainz.de/chartgallery/



(a) compare_execution_time_32 (b) compare_execution_time_64 (c) group_size_time_scaled (d) group_size_time_unscaled

Figure 3: Plots produced using the typical approach with default settings (Section 3.2).

3.3 Global Setup for Data Source and Sink

Using ChartGallery, let us now fix the aforementioned problems step by step *while* improving the quality of the plots. We start by changing how inputs and outputs are handled: Instead of having each plot manage this information individually and redundantly, we capture it *only once* in the global setup managed by ChartGallery. To do so, we first put the code of each plot generation into its own method. We then introduce the setup object from Section 2.1 (`Setup`) to define a named data source (`data_c` and `data_gs`) for each file. By simply passing this `Setup` to the `modify_setup` method, ChartGallery selectively overrides the default global setup with our new sources. Similarly, we pass parameters to the sink through a `SinkStyle` object, which sets the folder in which the plots should be materialized, another piece of information that is shared between all plots. To apply the global setup, we simply put the `@with_setup` decorator before each plotting method. To tap into a source for a specific plot, we add a parameter with the same name as the named source to the method. The default sink (included in ChartGallery) creates the output path from the user-set base path and the method name.

```
modify_setup(Setup(
  sources={
    "data_c": read_data("input-data/rtx-pq.csv"),
    "data_gs": read_data("input-data/rtx-group-size-scaling.csv"),
  },
  styles=[SinkStyle(base_path=Path("output-plots/"))],
))

@with_setup
def compare_execution_time_32(data_c):
  data_c = data_c.filter(pl.col("key_bits") == 32)
  data_c = data_c.select("misses_percent", "index_type", "VALUE")
  fig, ax = new_figure()
  bar_plot(data_c, ax, color_by="index_type")
  return fig
```

Code Listing 2: Sources and sinks in the global setup.

3.4 Global Setup For Plot Styles

So far, all four plots use the default style provided by ChartGallery. This style is sufficient to inspect the data, but re-uses all labels from the data source and auto-assigns colors, thus producing plots that are too rough for the final paper. Consequently, we want to format all plots to follow a consistent style (Figure 4). To do so, we simply modify the global setup again. We pass a `BarPlotStyle` for global settings. For column-specific settings, we pass multiple instances of `BarPlotColumnStyle`, which include the column title, entry-specific colors, hatches, human-readable names, and the preferred order of entries. Name translation can be done via a dictionary or a function. Note that styles can be re-used between figures, papers, and collaborators, so the number of style definitions remains mostly constant as the number of plots increases, and quickly amortizes the initial overhead. When a color or entry order needs to be changed afterwards, it can be done by changing a single line of code.

```
modify_setup(Setup(
  styles=[
    BarPlotStyle(apply_hatches_to_shades=True, base_color="white", ...),
    BarPlotColumnStyle("index_type",
      entry_colors={"rtx_index": "#d62728", ...},
      value_renames={"b_link_tree_16": "B+ Tree", ...},
      sorting_order=["rtx_index", "warpcore_80", "b_link_tree_16", ...],
    ),
    BarPlotColumnStyle("group_size_log",
      entry_colors={2: "#17becf", 4: "#ff7f0e", 5: "#2ca02c", ...},
      sorting_order=ASC,
      value_renames=lambda size_log: "group size " + str(2 ** size_log),
    ),
    BarPlotColumnStyle("indexing_method",
      entry_hatches={"2bvh": "", "1bvh_opt": "///"},
      value_renames={"2bvh": "naive impl.", "1bvh_opt": "opt. impl."},
      sorting_order=["2bvh", "1bvh_opt"],
    ),
    BarPlotColumnStyle("large_keys",
      title="Key size",
      value_renames={"0": "32bit", "1": "64bit"},
      sorting_order=ASC,
    ), # [further definitions omitted]
  ],
))
```

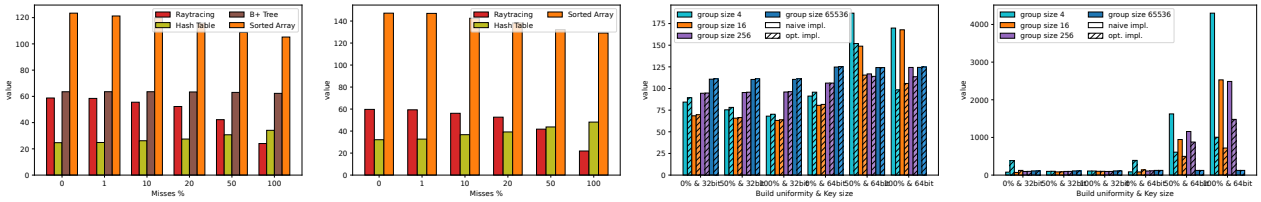
Code Listing 3: Defining plot styles in the global setup.

3.5 Introducing and Combining Group Setups

The attentive reader might have noticed that the plots in Figure 4 still have potential for improvement. First, the y-label of all plots still says "value" instead of "Time (ms)". Second, the x-ticks of Figures 4c and 4d overlap each other. Third, Figure 4b contains only three bars instead of four (there are no 64-bit results for the B+Tree) and hence, the plot can be narrower to save space. Obviously, we cannot define any of these settings globally, as they apply only to specific plots. To address this, ChartGallery allows applying setups to individual plots directly, which then selectively replace the global settings. To uniformly set the y-axis label, we create a corresponding setup object `time_grp` and apply it to all four plots by passing it to the `@with_setup` decorator. To create plots with rotated x-ticks, we define the setup `gs_grp` and pass it to the decorators of `group_size_time_unscaled/scaled`. Just like before, the operator << combines two setups, where the settings on the left take precedence. In a similar fashion, we define a `narrow_grp` to reflect narrower plots. Figure 5 shows two of the newly formatted plots.

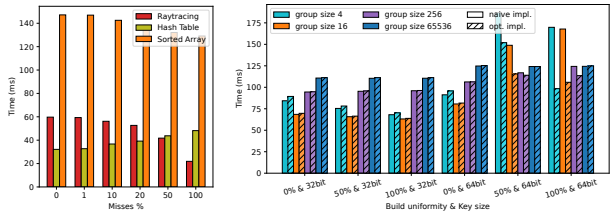
```
time_grp = BarPlotStyle(vertical_label="Time (ms)")
gs_grp = BarPlotStyle(horizontal_tick_rotation=15, legend_column_count=3)
narrow_grp = SinkStyle(width=4) << BarPlotStyle(legend_column_count=1)

@with_setup(time_grp)
def compare_execution_time_32(data_c):
  ...
@with_setup(time_grp << narrow_grp)
def compare_execution_time_64(data_c):
  ...
@with_setup(time_grp << gs_grp)
def group_size_time_unscaled(data_gs):
  ...
```

Code Listing 4: Individual group setups.



(a) compare_execution_time_32 (b) compare_execution_time_64 (c) group_size_time_scaled (d) group_size_time_unscaled

Figure 4: Camera-ready plots following a consistent and readable style (Section 3.4).



(a) comp_exec_time_64 (b) group_size_time_scaled

Figure 5: Selectively applying group setups (Section 3.5).

3.6 Variants

While the plots look presentable now, we can still find code redundancy within both pairs of plots due to their similarity. To finally address this problem, we use the *Variants* feature of ChartGallery. Listing 5 demonstrates this feature for the two plots that differ only in `key_bits`. Instead of having two distinct methods `compare_execution_time_32` and `compare_execution_time_64` containing redundancy, we create a single method that contains only the shared code of the former methods. The few differences, namely whether `key_bits` is 32 or 64, and the fact that the 64-bit plot is narrower, are expressed as two different setups. These setups are passed to the `variants` argument of a new `Setup`, which is then selectively applied to the `compare_execution_time` method. This change leads to ChartGallery automatically executing the method twice (once for each variant) and thereby creating two distinct plots. Note that the contextual parameter `key_bits` can be retrieved by declaring it as a function argument, similar to data sources.

```
key_bits_variants = Setup(
  variants=[
    Setup(parameters={"key_bits": 32}),
    Setup(parameters={"key_bits": 64}) << narrow_grp
  ]
) << SinkStyle(file_name="{meta_function_name}_{key_bits}")

@with_setup(time_group << key_bits_variants)
def compare_execution_time(data_c, key_bits):
  data_c = data_c.filter(pl.col("key_bits") == key_bits)
  data_c = data_c.select("misses_percent", "index_type", "VALUE")
  fig, ax = new_figure()
  bar_plot(data_c, ax, color_by="index_type")
  return fig
```

Code Listing 5: Using variants to avoid code redundancy.

3.7 Flexibility, Extensibility, and Accessibility

Note that despite focusing on consistency, ChartGallery does not fall short in terms of flexibility. The user can freely wrap arbitrary new plot types from various libraries, and parameterize them with their own plot style. One can also get creative with the fact that there is no type restriction for most objects: For example, it is possible to sink a table, and then create the plot in the sink, so that the core plotting function reduces to a single DataFrame query. Listing 6 shows the generation of the plot `compare_execution_time_32` as a line plot by using a different sink (`line_plot_from_table`) and by querying the source using SQL.

```
time_group = LinePlotStyle(vertical_label="Time (ms)")
sql_line_group = Setup(
  sink=line.line_plot_from_table(export_figures_sink)
)

@with_setup(time_group << sql_line_group)
def compare_execution_time_32_line(data_c):
  return data_c.sql("""
    SELECT misses_percent, index_type as index_type_LINES, VALUE
    FROM self WHERE key_bits = 32""")
```

Code Listing 6: ChartGallery is flexible regarding different plot types and querying languages.

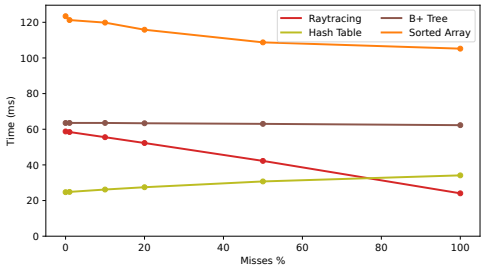


Figure 6: Line plot produced via SQL (Section 3.7).

4 CONFERENCE EXHIBIT

At the conference, we will present a code editor where visitors will be able to compare two scripts, one following the traditional plotting workflow, and the other one using ChartGallery. Visitors will also be able to experiment with ChartGallery: They can alter existing setup objects and see the changes propagate to all affected plots, and add arbitrary new setup objects or plot variants to observe composability and reusability of setups. We hope that presenting ChartGallery will spark interesting discussions about the plotting workflow of other research groups, and provide us with inspiration for future features.

REFERENCES

[1] Justus Henneberg, Felix Schuhknecht, Rosina Kharal, and Trevor Brown. 2025. More Bang For Your Buck(et): Fast and Space-efficient Hardware-accelerated Coarse-granular Indexing on GPUs. In *41th IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong SAR, China, May 19-23, 2025*. IEEE.

[2] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[3] Hassan Kibirige. 2024. *plotnine: A Grammar of Graphics for Python.* https://plotnine.readthedocs.io/ Accessed: 10 December 2024.

[4] Michael L. Waskom. 2021. seaborn: statistical data visualization. *Journal of Open Source Software* 6, 60 (2021), 3021. https://doi.org/10.21105/joss.03021

[5] Thomas Williams and Colin Kelley. 2024. *Gnuplot: An Interactive Plotting Program.* http://www.gnuplot.info