plotly

Star 23,447

Dash Python > ***Performance***

> Plotly Studio: Transform any dataset into an interactive data application in minutes with AI. **Sign up for early access now.**

Performance

This chapter contains several recommendations for improving the performance of your dash apps.

The main performance limitation of dash apps is likely the callbacks in the application code itself. If you can speed up your callbacks, your app will feel snappier.

Memoization

Since Dash's callbacks are functional in nature (they don't contain any state), it's easy to add memoization caching. Memoization stores the results of a function after it is called and re-uses the result if the function is called with the same arguments.

> For a simple example of using memoization in a Dash app to improve performance, see the "Improving performance with memoization" section in the **advanced callbacks** chapter.

Dash apps are frequently deployed across multiple processes or threads. In these cases, each process or thread contains its own memory, it doesn't share memory across instances. This means that if we were to use `lru_cache`, our cached results might not be shared across sessions.

You can add memoization caching with **Background Callback Caching**. Background callbacks are a great option for caching callbacks. In some cases, however, it may be more memory efficient to cache within your own functions that the callback calls, and not on the callback itself. Caching a function can give you more control over the specific functionality you want to cache.

Another option for caching is the **Flask-Caching** library, which saves the results in a shared memory database like Redis or as a file on your filesystem. Flask-Caching also has other nice features like time-based expiry. Time-based expiry is helpful if you want to update your data (clear your cache) every hour or every day.

Here is an example of `Flask-Caching` with Redis:

```
from dash import Dash, dcc, html, Input, Output, callback

import datetime
import os

from flask_caching import Cache

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)
cache = Cache(app.server, config={
    # try 'filesystem' if you don't want to setup redis
    'CACHE_TYPE': 'redis',
    'CACHE_REDIS_URL': os.environ.get('REDIS_URL', '')
})
app.config.suppress_callback_exceptions = True

timeout = 20
app.layout = html.Div([
    html.Div(id='flask-cache-memoized-children'),
    dcc.RadioItems(
        [f"Option {i}" for i in range(1, 4)],
        'Option 1',
```

```
        id='flask-cache-memoized-dropdown'
    ),
    html.Div(f'Results are cached for {timeout} seconds')
])


@callback(
    Output('flask-cache-memoized-children', 'children'),
    Input('flask-cache-memoized-dropdown', 'value'))
@cache.memoize(timeout=timeout)  # in seconds
def render(value):
    current_time = datetime.datetime.now().strftime('%H:%M:%S')
    return f'Selected "{value}" at "{current_time}"'


if __name__ == '__main__':
    app.run(debug=True)
```

Here is an example that **caches a dataset** instead of a callback. It uses the FileSystem cache, saving the cached results to the filesystem.

This approach works well if there is one dataset that is used to update several callbacks.

```
from dash import Dash, html, dcc, Input, Output, callback

import datetime as dt

import numpy as np
import pandas as pd
from flask_caching import Cache

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)
cache = Cache(app.server, config={
    'CACHE_TYPE': 'filesystem',
    'CACHE_DIR': 'cache-directory'
})

TIMEOUT = 60

@cache.memoize(timeout=TIMEOUT)
def query_data():
    # This could be an expensive data querying step
    np.random.seed(0)  # no-display
    df = pd.DataFrame(
        np.random.randint(0, 100, size=(100, 4)),
        columns=list('ABCD')
    )
    now = dt.datetime.now()
    df['time'] = [now - dt.timedelta(seconds=5*i) for i in range(100)]
    return df.to_json(date_format='iso', orient='split')


def dataframe():
    return pd.read_json(query_data(), orient='split')

app.layout = html.Div([
    html.Div('Data was updated within the last {} seconds'.format(TIMEOUT)),
    dcc.Dropdown(dataframe().columns, 'A', id='live-dropdown'),
    dcc.Graph(id='live-graph')
])


@callback(Output('live-graph', 'figure'),
              Input('live-dropdown', 'value'))
def update_live_graph(value):
    df = dataframe()
    now = dt.datetime.now()
    return {
        'data': [{
            'x': df['time'],
            'y': df[value],
```

```
            'line': {
                'width': 1,
```

Graphs

Plotly.js is pretty fast out of the box.

Most plotly charts are rendered with SVG. This provides crisp rendering, publication-quality image export, and wide browser support. Unfortunately, rendering graphics in SVG can be slow for large datasets (like those with more than 15k points). To overcome this limitation, plotly.js has WebGL alternatives to some chart types. WebGL uses the GPU to render graphics.

The high performance, WebGL alternatives include:

- `scattergl`: A webgl implementation of the `scatter` chart type. **Examples**, **reference**
- `pointcloud`: A lightweight version of `scattergl` with limited customizability but even faster rendering. **Reference**
- `heatmapgl`: A webgl implementation of the `heatmap` chart type. **Reference**

Currently, Dash redraws the entire graph on update using the `plotly.js` `newPlot` call. The performance of updating a chart could be improved considerably by introducing `restyle` calls into this logic. If you or your company would like to sponsor this work, **get in touch**.

Clientside Callbacks

Clientside callbacks execute your code in the client in JavaScript rather than on the server in Python.

Read more about clientside callbacks in the **clientside callbacks** chapter.

Background Callbacks

In addition to providing memoization caching, background callbacks can help you improve your app scalability by moving computations from the Dash app server to a background job queue. See the **Background Callbacks chapter** for more information on how to implement these.

Partial Property Updates

With **partial property updates**, introduced in Dash 2.9, you can improve your app performance by only updating the parts of a property that you want to change. For example, the color of data points on a graph, but not the underlying data. See the **partial property updates** page for more information.

Data Serialization

New with Dash 2.0, you can use `orjson` to speed up serialization to JSON and in turn improve your callback performance. For an app that runs complex callbacks involving a large amount of data points, you could see a performance increase of up to 750 ms. **Learn more about `orjson` here** or install it with `pip install orjson`.

`orjson` is completely optional: If it exists in the environment, then Dash will use it; if not, Dash will use the default `json` library.

Sponsoring Performance Enhancements

There are many other ways that we can improve the performance of dash apps, like caching front-end requests, pre-filling the cache, improving plotly.js's webgl capabilities, reducing JavaScript bundle sizes, and more.

Historically, many of these performance related features have been funded through company sponsorship. If you or your company would like to sponsor these types of enhancements, **please get in touch**, we'd love to help.