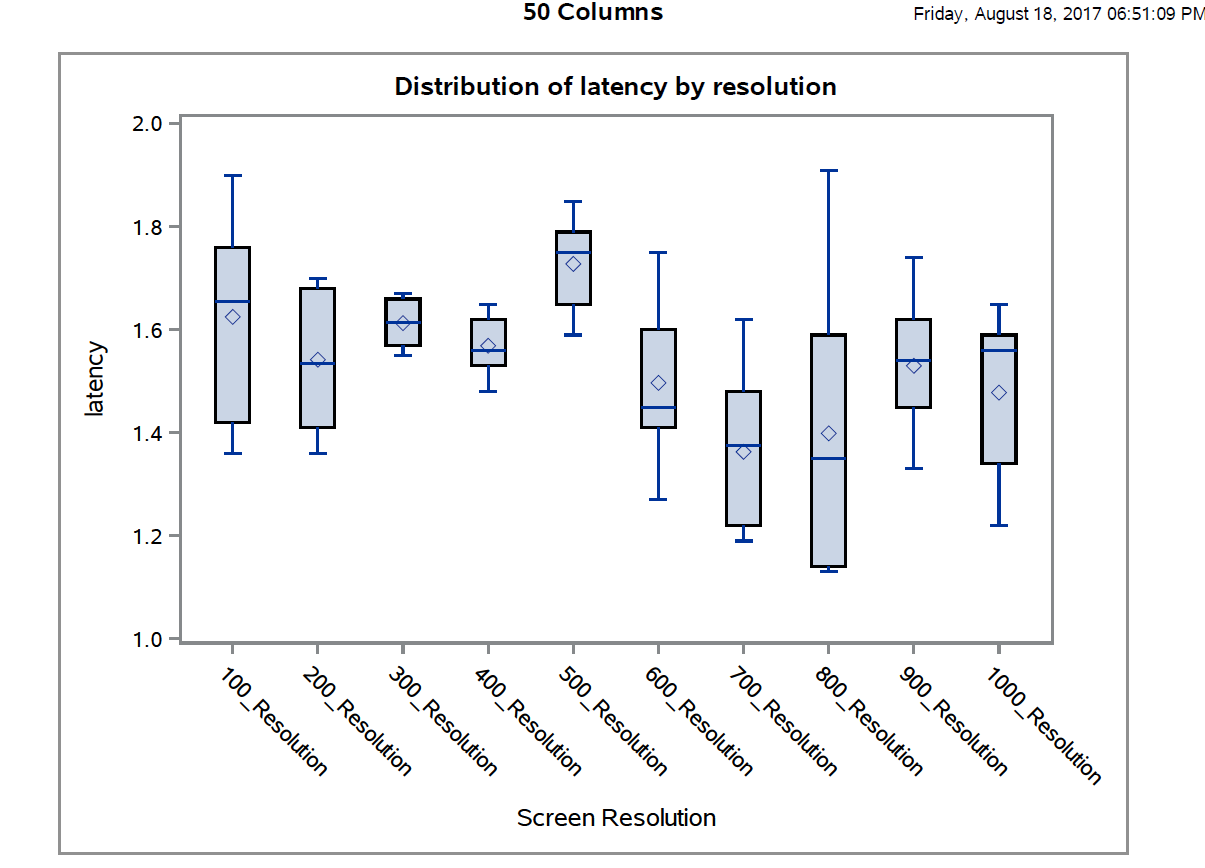
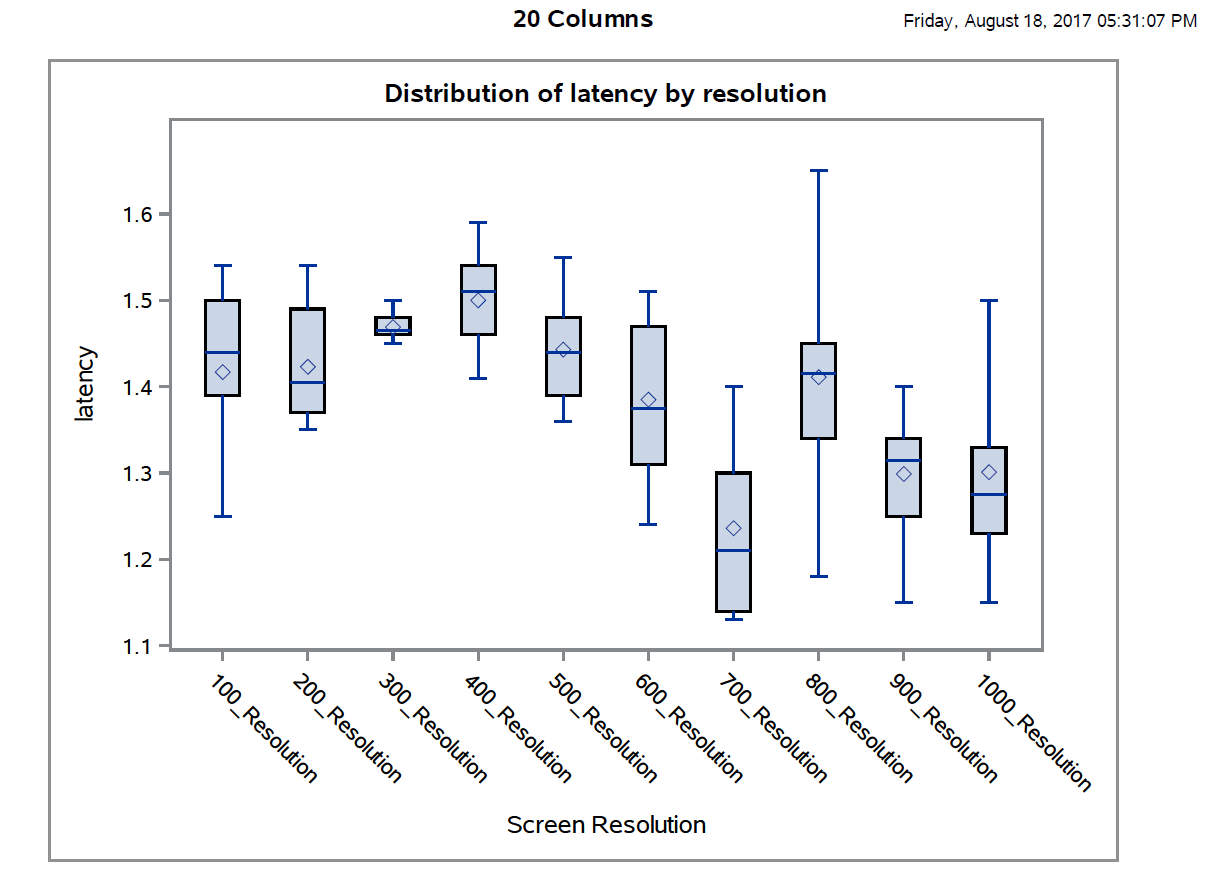
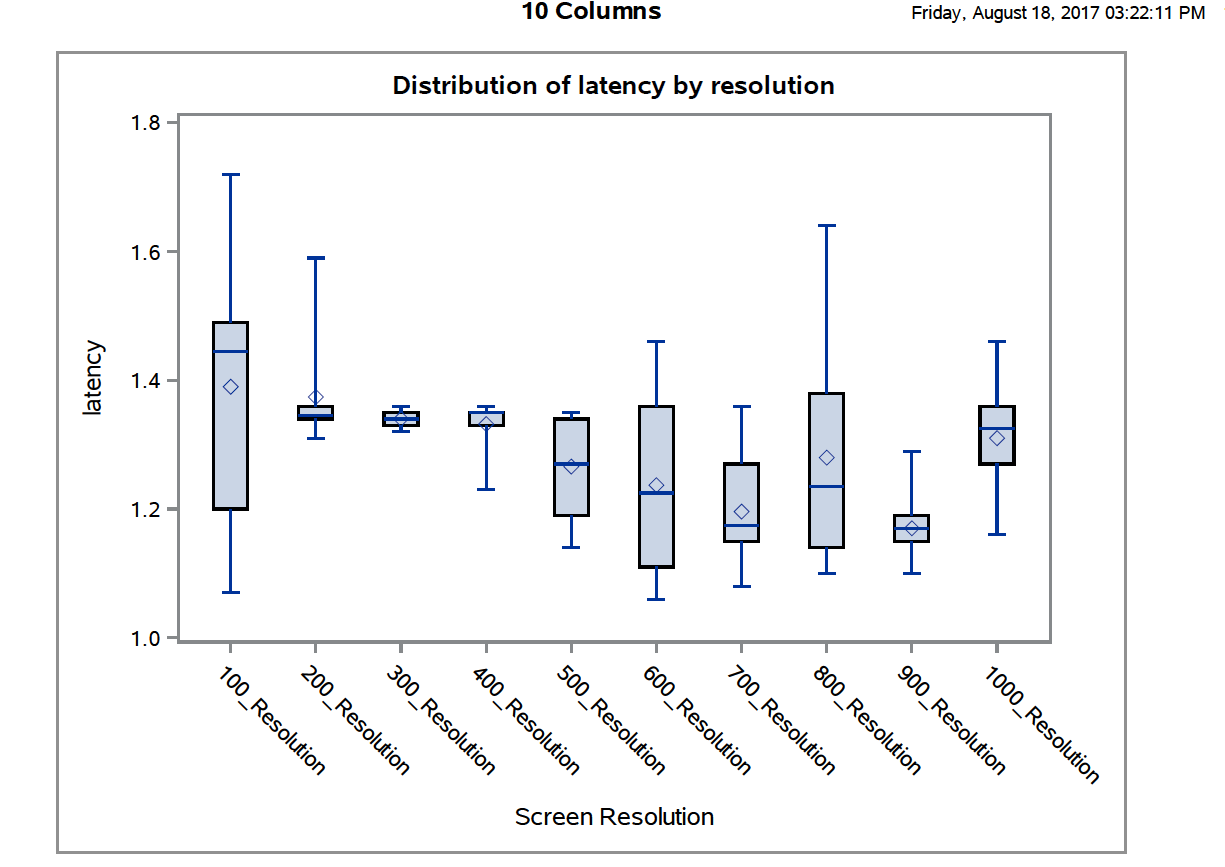
Shawn Idahosa

CMSC499A

Epiviz-py backend Data Scaling Report

8/16/17

Epiviz is a computational and visual data analysis of genomics; workflows that involve the integration and exploration of multiple heterogeneous data sources, small and large, public and user specific have been poorly addressed by these tools. Epiviz wants to fix that and make it simpler for scientists to use for their own research on studying genomes. The general concept of the research project is to push more of the data analysis supporting interactive visualization to the backend. The system will display visualization resource constrained data transformations and different visualizations that the data scaling will support include scatter plots, heat maps, stacked bar charts and other graphics to help clients organize data. Before I could start doing anything to the backend of the Epiviz, I learned how to make a web app using Flask as the framework, learning the inner works of MySQL, and Pandas data-frame for organizing the data. Learning these new technologies and putting them into practice was the fun part of the research, I walk away with a better understanding of how databases work, data frames purposes, and how to structure a web app. With the data being scaled and transformed in the backend, there was an important question that needs to be addressed. By pushing the query to the backend instead of the front end, would it cut latency?

Constructing the first task didn’t take long; the most important part of this task was to learn Flask and MySQL. The next couple of tasks was practicing with Flask and MySQL, learning the syntax of MySQL, and preparing the logic for the last task, the scale\_data function. Scaling the data takes in a chromosome, a start and end codons, resolution aka the max number of pixels the graph/chart will be, and columns the client wants to see charted. Once you query the data, if the number of rows of query data is bigger than the resolution number by a significant amount, the function gets the average for each column every “resolution” number. For example, if the number rows of data are 2000 and the resolution is only 100, for every 20 rows of data, the average would be taken until we have 100 rows of results to fit in that resolution. Using pandas, I was able to do this step in one line of code, “df = df.groupby(df.index//(len(df)/resolution).mean()” (df being the query data, len(df)/resolution getting me the number of rows I want to average out every time). My next task was to get the latency running the web app on the server to find out if the querying the data on the back-end makes a significant difference than doing it on the front-end. To measure latency, I used a tool called wrk that will run once the web app server is online; it will then make a few connections and threads to the web app and display the results. 

When inputting a resolution size of 300, it creates the smallest range of latency values for all three tests while inputting a resolution of 800 has the largest range. There is always a slight increase in latency for a resolution of 400-500, but the graph shows it takes a massive dive after 500. The average for all three tests would be a latency of 1.4 seconds (looking at the graph line plots). The line graph shows that the latency decreases as the resolution size increases, but starts high if the number of columns inputted is a large number.

Scaling data is much easier to do with a larger resolution like 1000 for a query of 2000 as opposed to a resolution of 100. The function is trying to fit large amounts of data into one pixel in a small resolution size, so it will take longer to trim down the query to fit the desired resolution. A large resolution number inputted doesn’t need to scale down the data so much, thus the process finishes faster and the latency is lower. The data from all three tests shows the latency time increases as the number of columns inputted increases, which can essentially boil down to this statement, the larger the data, the higher the latency time is. For the entire column test, it shows as the resolution size increases, the latency time decreases. This is caused by that fact it takes more time for the function to get fit huge amounts of data in such small resolution sizes. The backend sends a request to the MySQL database, wants to retrieve back a copious amount of data to be sent to the data frame, then be transformed and finally send a response. The front end, however, would have to make a request to run the backend first to run the same logic to retrieve the data. Along with that, the front also has to format the data to JSON to be able to use it. My backend code can make the request and return a ready to use format in less time than a front end version would.