DREU: Final Report

Hyekang Joo

August 3, 2021

1 Introduction

Initially produced by a team[1] at the University of Washington, Vega is "a declarative format for creating, saving, and sharing visualization designs"[2]. This summer, I was invited by Dr. Leilani Battle, who holds a joint appointment at the University of Washington and the University of Maryland, to assist the professor and her graduate students with an ongoing project. The project is an enhancement of the former project Voyager [2], with the focus on Vega optimization. Through Vega optimization, the team aims to develop it into a scalable product, allowing data of significantly large size to be loaded and queried at a faster pace than by others.

Vega is the core of the project as it is the visualization language used in the project to make the scalability possible. Vega is a powerful tool as it has the capacity to render a visualization with a JavsScript Object Notation (JSON) file, as opposed to other visualization languages such as D3, which runs on a line-by-line basis like ordinary programming languages. As a result, Vega offers a lot of advantages over those, such as a write-up simplicity and an ease of understanding the code and visualizing the output just from looking over the file. However, as the datasets continue to grow, Vega needs the ability to scale further; Vega's data loading and processing process typically runs in the browser, leading to its scalability limitations. First, the amount of data to visualize is restricted by the browser's memory. Second, all the computation load for processing data like filtering and aggregation is placed on the client machine that runs the browser application. In this project, we want to address Vega's scalability limitations through a series of optimization.

My main contributions to the project during the research internship over the summer can be summarized in two points: upgrading the outdated packages to the latest version and running benchmark tests to pinpoint the elements of the data processing workflow that can potentially be optimized.

[1] https://github.com/vega/vega
[2] https://vega.github.io/

2 Related Work

2.1 Multiple Comparisons Problem

According to Zgraggen et al., the Multiple Comparisons Problem (MCP) is prevalent in the world of visualization [3]. As a matter of fact, the more an analyst views visualizations, the more likely he/she is to encounter a false discovery (Type I Error) or false omission (Type II Error) [3]. In the paper, Zgraggen et al. demonstrates the importance of running a confirmatory data analysis on a separate validation dataset and the grave danger of running it on the same dataset. Therefore, we intend to use a separate dataset as a validation dataset in lieu of our original dataset to combat MCP.

2.2 Dataset Generation

Battle et al. [1] conducted a study in 2020 to answer the question, "Are database systems truly capable of supporting real-time interactive data exploration at scale," where "real-time" is defined as having a latency of sub-100 milliseconds or supporting at least 10 frames per second. In the process, the team created a dataset generator because they needed "the ability to scale the data up to measure effects on Database Management System (DBMS) performance" [1]. Their dataset generator generates new samples based on the statistical models the generator creates, which captures attribute relationship and patterns of the original dataset. Their application was tested on nine dataset cases (synthetic flights, movies, weather data) $\times$ (1M, 10M, 100M rows) to illustrate broader performance patterns as the data is scaled up. In our study, we borrowed the dataset generator that was used in the study by Battle et al.

3 Package Upgrade

Because Voyager [2] was produced in 2017, the project required massive package upgrades for it to be up to date. Upgrading packages was a strenuous and repetitive task because it required me to recursively check and keep note of the peer dependencies of the package I am trying to upgrade. In addition, I had to check the CHANGELOG or release notes for changes between versions and make changes accordingly to ensure a safe upgrade without any inconsistency. While it was a bit taxing at first, it was also rewarding as I was able to not only learn to upgrade packages, but also communicate with Dr. Dominik Moritz, a co-author of Voyager, during the code review and ultimately get two of my pull requests approved of by him for integration. Out of about 100 outdated packages it has, I contributed to upgrading about 40 packages to the later version, more than half of which have been upgraded to the latest version. While there are a lot more to go, I am glad that a large number of packages have been successfully upgraded and pushed to the main Voyager repository[3]. More detailed information about

[3] https://github.com/vega/voyager

my work can be found at Pull Request #858 and #860 in the repository.

4 Benchmarking Pipeline

In this project, Vega optimization is one of the core objectives. Through the optimization, we intend to make a more scalable program. By creating a scalable program, it should be able to process a large amount of data at the cost of a fraction of its overall performance. In other words, as the dataset size increases, the speed at which Vega renders the corresponding visualizations should gracefully degrade, or not degrade at all. In the process of optimization, running a micro-benchmark – benchmarking the performance of *some* of our tool's features against that of other languages' – is a key step because we need to empirically prove to the audience that our Vega-integrated, developed work outperforms other state-of-the-art approaches. Thus, in this section, I will be discussing the project's micro-benchmarking procedure.

4.1 Data Generation

First, in order to run the test, it is necessary to have a dataset of large size to prove the scalability of the work. As the objective of the project is to allow the original product to be able to hold data of larger size without much degradation in performance (while performing better than its competitors if possible), we prepared large datasets for the experiment. Rather than downloading a pre-existing massive dataset, we decided to generate a big dataset by feeding a small dataset to a data generator. We opted to generate a dataset for two reasons. First, through the usage of a dataset generator, we are able to control the output size, which provides us with the generated dataset's size flexibility. Second, it is important to conduct confirmatory analysis on a dataset different from the original dataset to combat MCP [3]. For the data generation, we used the dataset generator created by Battle et al. in the Crossfilter Benchmark [1] rather than our own generator because its usage fits in our work and it is good to have an established baseline that the community considers a good starting point. The generator requires the following three inputs: 1) data in a CSV format; 2) metadata listing the type of each attribute (quantitative or qualitative) in a JSON format; 3) seed value. The data generator starts off by converting the quantitative data into z-scores and collecting the covariance of the data. Then, in the process of data generation, it converts each z-normalized data into the corresponding value of the normal CDF and applies the inverse CDF to the correlated random data. Terminally, the generator de-normalizes and produces a dataset of size N with the normal distribution.

For convenience, I created a bash script that allows the user to easily generate data by specifying the path to the aforementioned three inputs and to an output (generated dataset). Everything is available in the GitHub repository[4] I created,

[4]https://github.com/joos2010kj/voyager-microbenchmarker/tree/kevin

and the repository also contains a sample dataset (cars.csv)[5] that the user can test should they not have a dataset to test with. This has been tested with multiple datasets (weather, movies, cars, flights) for assurance, and they all were successfully able to generate a new dataset comprised of millions of sample data as specified.

4.2 Vega

4.2.1 Discussion

Microbenchmark is a form of measuring and testing the performance of a single component or task. Microbenchmarks are useful for estimating the performace of the atomic components of Vega dataflows (and their corresponding SQL queries). With an understanding of how each component performs, as well as the performance relationships between components, we can estimate the cost of previously unseen data flows, a critical feature of standard database optimizers. In the experiment, we conducted a few microbenchmarking tests by measuring the performance of multiple operators individually to understand how Vega compares to other query languages as well as how Vega works for the partitioning process in the future. In addition, we collected the statistical information such as run time and conducted cost-based planning. We tested the operators individually to empirically show the performance of its own to the audience. The ones listed below are the operators that perform data transformations of the most commonly used visualization types. They can be expressed in common SQL statements as well, which is necessary for the performance comparison via micro-benchmarking. After deliberation, we arrived at the following conclusion: Our approach to address the scalability limitation is to offload resource-intensive operations to a separate DBMS. With the unprocessed large datasets stored in a DBMS, common data processing operations such as aggregation, filtering and projecting can screen and reduce the size of data to store in the client machine for visualization. In this case, we can deal with datasets that would otherwise be too large to load and avoid wasting space for unnecessary data in the application. Meanwhile, operations such as extent, bin, stack and collect can be combined with the previously mentioned operators to transfer their computation load for sorting, partitioning or grouping from the client to the DBMS. For these reasons, we focus on Vega operators with equivalent DBMS operators in our microbenchmarks.

1. Extent: Compute the minimum and maximum values of the data
2. Stack: Find the type of stacking offset if a field should be represented or shown in a stacked manner
3. Filter: Cherry-pick only those that meet the specified condition from the data

[5] https://vega.github.io/vega-datasets/data/cars.json

   (a) Different Vega filter predicate types
   (b) Different filter values and ranges
4. Aggregate: Group the data or compute the summary statistics over groups of data
   (a) Type of operations (sum, valid, median, etc.)
   (b) Groupby fields (how many buckets, what are the sizes of each buckets)
   (c) Type of variable (str, number, quantitative vs. categorical, etc.)
5. Bin: Put the quantitative data in discrete bins with a specific range
   (a) Number of bins applied
   (b) Different bin ranges
6. Collect: Sort the data in ascending or descending order
   (a) Number of fields in the sort parameter
7. Project: Perform a relational algebra projection operation
   (a) Number of fields selected

4.2.2 JavaScript to Vega

In order to run the benchmarking test (Vega vs. PostgreSQL vs. DuckDB) through Node.js, an auxiliary tool is needed. The core functionality of the tool should be the generation of random query commands (in Vega, PostgreSQL, and DuckDB) that utilizes the aforementioned seven operators in each language for benchmarking purposes. Therefore, I assisted the professor with creating the generator for Vega, while my mentor (fellow graduate student) handled producing the generator for the other two languages.

The process of generation involves two parts: constructing a random **method (operation)** generator for the **operator** and creating a random yet appropriate **argument** generator for the method. To elaborate on the concept of "method," each operator has unique methods bound to itself, and an operator always functions in combination with at least one of those methods; those methods, or operations, can be visualized as the subcategories (e.g., minimum, maximum, mean) of an operator (e.g., aggregate) that make the operator complete. Refer to Figure 2 for an example. As for the argument, since each method may require different types of input (parameters), it requires an appropriate argument generator as well. More details about the argument generation process are written in the next subsection.

First, I worked on creating a class that converts the query commands into Vega query command for Extent, Filter, Aggregate, Collect, and Project operators. A sample conversion can be found in Fig 1 and Fig 2 below.

```
Transform.Aggregate.Merge(
    Transform.Aggregate.valid("foo"),
    Transform.Aggregate.min("bar"),
    Transform.Aggregate.groupby("sample")
);
```

Figure 1: Sample JavaScript query: Transform.Aggregate.Merge function combines all the operations into one and returns a Vega query command. Refer to Fig 2 for its output.

```
{
  "type": "aggregate",
  "fields": [ "foo", "bar" ],
  "ops": [ "valid", "min" ],
  "groupby": [ "sample" ]
}
```

Figure 2: Sample output in Vega format: This query is using the "aggregate" operator, and it is using the "valid" operation, which counts the number of "foo" values that are not missing (i.e., NaN) in the database, and the "min" operation, which finds the minimum "bar" value in the database (https://vega.github.io/vega/docs/transforms/aggregate/). In addition, the output is grouped by "sample" field.

4.2.3 Parameter Generation

I created a program that generates the parameters for each operator during the querying process so that the user does not have to manually specify the parameter, such as the range of values and the name of the attribute (i.e., generate a random yet appropriate argument for each function in Transform class as in Fig 1). However, because the user may not want an unrealistic parameters (e.g., querying for cars that weigh between 10 pounds and 20 pounds or between 0 pounds and 1 million pounds), I also collected mean, standard deviation, minimum, maximum, integer/float, etc. to generate the parameters based on the normal distribution (independently) from the seed datasets for a more desirable outcome. We automate the parameter generation to (a) minimizes manual errors and (b) ensures that the parameters are generated in a consistent way from realistic inputs.

Fortunately, the dataset generator from Crossfilter Benchmark [1] already had some script for getting mean and standard deviation from an input dataset, so I borrowed a portion of their code to reduce the duplication of research works and to ensure that our evaluation methods are consistent with prior research.

	mean	stdev	min	max	type
Name	N/A	N/A	N/A	N/A	categorical
Miles_per_Gallon	23.445	7.805	9	46.6	float64
Cylinders	5.471	1.705	3	8	float64
Displacement	194.412	104.644	68	455	float64
Horsepower	104.469	38.491	46	230	float64
Weight_in_lbs	2977.580	849.403	1613	5140	float64
Acceleration	15.541	2.758	8	24.8	float64
Year	N/A	N/A	N/A	N/A	categorical
Origin	N/A	N/A	N/A	N/A	categorical

Table 1: Sample metadata of a generated data, to be used for parameter generation

4.3 Benchmarking

All micro-benchmarking experiments were conducted on a single server (Red Hat 7 operating system), with 20 GB of memory, 12 cores (Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz), and over 30 TB of disk space. We ran the test with the four real-world datasets cars, flights, movies and weaterh dataset[6] of size 500,000, 1 million, 2 million, 5 million and 10 million, all of which were generated using a dataset generator (Refer to Section 4.1 for more information). For your convenience, we plan to share a GitHub repository that helps a user to replicate our experiment step-by-step.

Using a fellow teammate's code as a template, I ran the vega query test for extent, collect, and project, while the teammate ran the test for aggregate, bin, filter, and stack. While it requires further investigation for confirmation, currently, Vega's runtime for querying turned out to be generally very good in transform tests in comparison to PostgreSQL and DuckDB. For example, while Vega was marginally worse than PostgreSQL in aggregate, it was better than the runner-up with typically a bigger gap in the tests with bin, collect, filter, and stack. Although it did not claim the first place in aggregate, extent, and project, Vega managed to beat the competitors the most in the fourteen tests conducted.

To validate our result, we plan to investigate into this by running the experiment with datasets of more different sizes.

5 Conclusion

In terms of individual operators, operations through DBMS are significant faster than Vega with larger datasets except for the stack operation for its complicated query. For operations that return smaller results like aggregate and extent, DBMS are always preferred and especially with large datasets. Through the

[6]https://vega.github.io/vega-datasets/data/cars.json

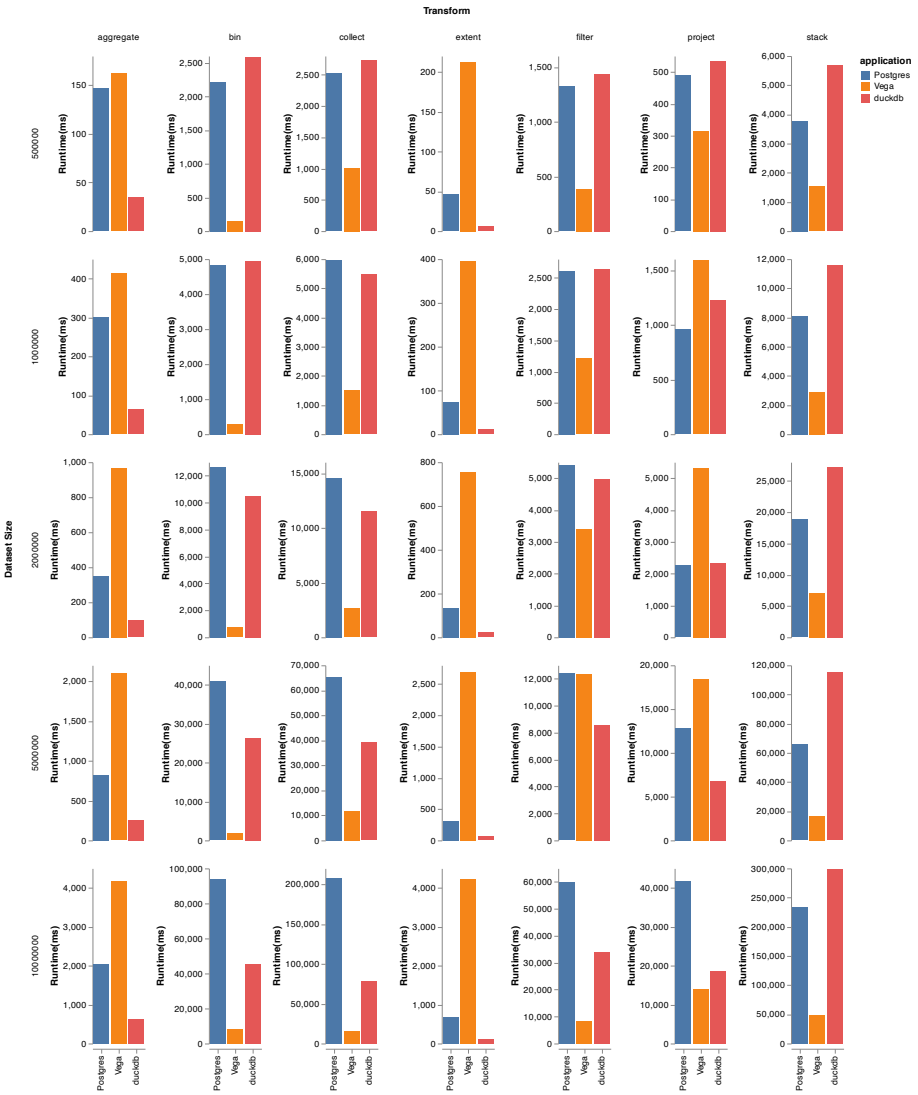

Figure 3: Bar chart runtime comparison among Vega, PostgreSQL, and DuckDB.

benchmarking process, we have come to find that Vega outperforms PostgreSQL and DuckDB in general even at larger dataset sizes. This result is promising because it suggests that there is not an obvious right answer to when queries should stay in Vega and when they should be offloaded to a DBMS, which in turn shows that our future optimizations would be useful for the database community to learn about.

6 Acknowledgements

Overall, I am very glad to have been a part of Dr. Battle's team this summer through CRA-DREU program. Through this, I have been able to learn a lot of technical materials in regards to software engineering, human-computing interaction, and database management. I find this internship invaluable because, as a rising first-year graduate student, I have been able to experience the life of a research assistant firsthand and gain the knowledge in software engineering and human-computing interaction, which are two areas of my interest, through the exposure. I hope that my contributions during the internship do not go to waste, but become a part of the final work. If an opportunity is given, I am more than happy to continue to assist Dr. Battle with the project after the end of the internship.

References

[1] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. Database benchmarking for supporting real-time interactive querying of large data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1571–1587, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager 2: Augmenting visual analysis with partial view specifications. In *ACM Human Factors in Computing Systems (CHI)*, 2017.

[3] Emanuel Zgraggen, Zheguang Zhao, Robert Zeleznik, and Tim Kraska. *Investigating the Effect of the Multiple Comparisons Problem in Visual Analysis*, page 1–12. Association for Computing Machinery, New York, NY, USA, 2018.