Contributions to Vega

Ryan Russell

June 9, 2016

A senior thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science with College Honors in Computer Science at University of Washington

Presented on 06/01/2016
Approved by advisor Jeffrey Heer

Signature: ________________________________

Date: ________________________________
### Contents

1 Abstract .............................................. 3

2 Background ......................................... 3
   2.1 The Dataflow Graph ............................. 4
   2.2 The Expression Language ...................... 4
   2.3 Streaming Data Transformations ............... 6

3 Consistency and Expressivity ......................... 6
   3.1 Replacing Predicates with Expressions ........ 7
      3.1.1 The \texttt{indata} and \texttt{inrange} functions .... 8
      3.1.2 The \texttt{scale} and \texttt{iscale} functions ......... 9
   3.2 Expression Syntax Extensions .................. 9

4 Performance ....................................... 10
   4.1 Dataflow Optimizations ....................... 10
      4.1.1 Automated Sorted Indexes ................. 10
      4.1.2 \texttt{prev\_update} Optimization .......... 12
   4.2 Rendering Optimizations ...................... 13
      4.2.1 Minimizing Canvas State Changes .......... 13
      4.2.2 Combining Canvas Paths .................... 14
      4.2.3 Evaluation .............................. 15

5 Future Work ..................................... 16
1 Abstract

Data visualization is important, and the browser is an increasingly important platform for viewing and interacting with data. Online journalism and analytics dashboards are just some applications where visualization tools must support richer interactivity and handle larger datasets than ever before.

Vega [2] [3] is a previous work which defines a grammar for describing interactive visualizations. The open-source implementation, written in JavaScript, parses the specification into a dataflow graph to efficiently render the visualization and the results of interaction in the browser, or on a server with Node.js. It is built to be expressive and performant.

Vega is a library for creating visualizations, but it is also a domain-specific declarative programming language. During my time at the Interactive Data Lab, I have contributed to several projects in the Vega ecosystem, including Vega\textsuperscript{a}, Lyra\textsuperscript{b}, and Datalib\textsuperscript{c}. This paper focuses on contributions to Vega as a programming language. I considered the language design concerns of consistency and expressivity as well as the implementation concern of performance.

2 Background

This section provides the necessary background on Vega to understand the contributions in this paper. More information can be found in the Vega papers [2] [3] and the documentation\textsuperscript{d} on Github.

Vega is a JavaScript library which takes a JSON specification for a visualization and renders it in the browser. The visualization can react to user input events and JavaScript API calls. Rather than giving a fixed set of chart types and interactive behaviors, Vega provides a set of primitives for creating novel chart types and interactions.

Marks are the main visual elements of the visualization's data. These include rectangles, textboxes, symbols, and lines. A mark may be singular, or it may have an instance for each data point in a data source. Its visual properties, such as position, color, and size, may depend on fields on its datum or global values such as signals, and may be transformed by a scale. Production rules are if-else chains which set the value of a visual property to one of several values depending on conditions. Predicates are definitions for selecting specific data points using comparison, logical, and other operators. Predicates can be used as the condition of a production rule or to trigger a data source to reset and clear its elements.

\textsuperscript{a}https://github.com/vega/vega
\textsuperscript{b}https://github.com/vega/lyra
\textsuperscript{c}https://github.com/vega/datalib
\textsuperscript{d}https://github.com/vega/vega/wiki
Signals are reactive variables that can parameterize the visualization. A signal can react to user input events using a CSS-like event selector syntax.

Scales are scaling functions which transform data-space values into visual-space encodings. The parameters of a scale — its domain and range, for example, can depend on the values in a data source or the value of a signal. A scale transform can also be inverted to change a visual-space value back into a data-space value. Axes and legends are visual representations of scales.

Data sources are collections of data points. The data may be loaded from a URL, defined inline, added with the API, or come from the output of another data source. Every data point is a JavaScript object with a set of fields. A unique ID is generated for each data point. Each data source has a pipeline of data transformation operators which may modify the data points. These include filter operations, derived fields, summary statistics, and complex visual layouts. Data points can be added, removed, or toggled in response to the changes of predicates or signals.

2.1 The Dataflow Graph

Vega creates an acyclic dataflow graph to handle updates. External changes like API calls or user input events can trigger an update. The dataflow graph nodes are sorted topologically, and a changeset object keeps track of every object that is changed — such as data points, signals, and scales. Each object in the dataflow graph will update if any of its dependencies have changed. In order to avoid circular dependencies, certain types of connections are not made. For example, a signal may transform its value using some scale, and that scale may in turn change its parameters based on that same signal. However, the scale will always depend on the signal and not vice-versa. Signals are updated before scales, so the signal will use the old scale function to determine its value, and then the scale change. Vega does not attempt to discover circular dependencies, only to ensure by the construction of the dataflow graph that none can occur. A signal will never depend on a scale, even if the dependency would not create a circular dependency.

2.2 The Expression Language

In many locations, Vega allows embedded JavaScript expressions. This can be used, for example, to filter a data source based on the result of an expression. The expression language is mostly a subset of JavaScript\(^4\). For security reasons and ease of static analysis, it only allows a limited set of side-effect free operations. In the expression language, all functions and signals are global values.

\(^4\)There is a global function with the reserved word name if which would not be allowed in standard JavaScript. It is transformed into a ternary operator expression at compile time.
1. Input Expression

\[
\text{clamp(eventX(), 0, width)}
\]

2. AST

```
{
  type: "CallExpression",
  callee: {type: "Identifier", name: "clamp"},
  arguments: [
    { type: "CallExpression",
      callee: {type: "Identifier", name: "eventX"},
      arguments: []
    },
    {type: "Literal", "value": 0, raw: "0"},
    {type: "Identifier", name: "width"}
  ]
}
```

3. Output Expression

\[
\text{Math.min(this.sig["width"].value, Math.max(0, event.vg.getX()))};
\]

Figure 1: Three stages of expression compilation. Note that the \texttt{clamp} function is a macro which uses \texttt{Math.max} and \texttt{Math.min} to implement its functionality, signals are not global values, and \texttt{eventX()} is renamed. Calling \texttt{event.vg.getX()} in the input expression would not be allowed, because of the function whitelist.

Method calls and function definitions are not allowed.

The implementation of this is more complicated than \texttt{eval}ing the expression. We parse the expression into an abstract syntax tree using a modified version of the Esprima parser\(^1\). Our modifications throw errors if disallowed pieces of syntax are used. In the syntax tree, we find which data source fields and signals are accessed to create dependencies in the dataflow graph. We convert the syntax tree back to a JavaScript expression. The generated expression has many changes. Signals, though they appear as global variables, are transformed to accesses on parts of Vega’s internal model. Function calls can act like macros, but most function calls are transformed into other function calls. For example, all of the Math.* functions are available as globals, but are transformed into references to the function on the global Math object. Other functions are injected as properties of the object \texttt{this.defs} in the generated function. See figure 1 for an example.

\(^1\)http://esprima.org/
2.3 Streaming Data Transformations

Many of Vega’s data source transformation operators work without keeping a complete copy of the data. They receive a changeset which includes a list of added, modified, and removed tuples, and pass the resulting changeset to the next operator. After all the operators are done, the last changeset is passed to a collector, which materializes the data into a complete set of points. Operators which need access to all data points at once use intermediate collectors.

As a concrete example, consider the filter data transformation. A filter includes a test expression which can depend on attributes of the datum or global values such as the value of a signal.

If a global dependency changes, then the filter operation must re-evaluate every data point and return a changeset to the next operator. However, streaming data changes can be handled more efficiently. Newly added data points are tested. If they pass the filter expression, then they are put in the output changeset. If not, the data point’s id is marked as skipped, and it is not added to the output changeset. If a data point is removed, it is ignored if it was marked as skipped, otherwise it is removed in the output changeset. Modified data points use the result of the filter expression and whether it is marked as skipped to determine whether to put the point in the added, removed, or modified changeset lists.

3 Consistency and Expressivity

As a programming language, Vega uses several pieces in combination — signals, marks, data sources, etc. — to create visualizations. Consistency in the language design requires orthogonality. Each piece should be sufficiently different to justify its inclusion in the language, and they should combine to provide a full feature set. Predicates and filter expressions were two features that provided the same functionality in different contexts. Predicates had only one function, but expressions could be used for many purposes. I made changes to expressions so that they could completely replace predicates and predicates could be deprecated.

Another important language design concern is expressivity. Given the small set of basic language features, the language should be able to express a large variety of programs. Vega is a domain-specific language, and I was not interested in extending it past its role as a visualization language, but I worked to increase the expressivity of the language to make common visualization tasks easier to express.
3.1 Replacing Predicates with Expressions

Predicates, which have since been deprecated, were a top-level object in Vega. A predicate takes input arguments and the values of signals, and returns a boolean value. For example, an "in" predicate may check if a value is inside of a given range, or if the value exists as a property of a data point in some data source. A production rule could use a predicate to change the visual properties of a mark instance. For example, predicates could check whether a point is in a selected region of a visualization, and then points could be styled if they are selected.

A predicate used in a production rule is similar to a filtering operation. Vega also supports filtering the output of a data source. However, predicates and filter transforms used very different syntax. Here are two examples that can be used to select the same set of data points. The first is a predicate definition, and the second is a filter expression.

```
1
2   {  
3     "name": "inStock",  
4     "type": ">",  
5     "operands": [{"arg": "count"}, {"value": 0}]  
6 }  
7
1
2   {  
3     "type": "filter",  
4     "test": "datum.count > 0"  
5 }
```

Filter transforms are defined using the Vega expression language, but predicates were defined with an AST-like syntax. The syntax did not support complex expressions, but multiple predicates could be combined with `and` and `or` operators. The syntax did not support all of the JavaScript operators — only comparison and boolean logic operators, while expressions allow arbitrarily complex expressions with most of the JavaScript operators.

Predicates also had features that expressions did not. They used the AST syntax to ease static analysis and allow more complex queries. One query which expressions could not match was the `in` operator\(^9\). On a range, the `in` operator simply checked that a value was in a specific range, which could be emulated using `max`, `min`, and inequality operators, but `in` could also query inclusion in a data source. For example, the predicate "selected" (figure 2) would return true for a data point if there is a data point in the data source "selectedPoints" with the field "_id" equal to the argument "id". This feature, combined with modifiable data sources, allowed complex interactions like multiple-select to be defined declaratively. Predicates could also apply scale transformations and inversions to their arguments, while expressions could not. In order to replace

\(^9\)Not related to the JavaScript operator `in`, which checks if a key exists on an object.
predicates with expressions, a comparable feature for each of these was added to the expression language.

3.1.1 The indata and inrange functions

To implement the predicate in operator, the expression language now has the functions inrange and indata. inrange simply checks whether a value is between the bounds given. This is simply a convenience feature, to match the predicate in operator on a range. indata is more complicated. indata takes a data source name, a value, and a field to lookup. indata returns true if there exists a data point in the named data source with the given value as the value of the field.

The first argument to indata is the data source name. This must be a literal string. This is because the data flow graph does not support a link from an unknown data source. Functions that are defined in the expression language are implemented as macros, with JavaScript code transforming the function arguments into code at compile time. The code generator function for indata checks that the first argument is a literal string, and throws an error if it is not.

indata is a constant time operation. To achieve this, we dynamically create indexes on the data source. The index is like a histogram. It maps each value that appears on the field to a count. Like the rest of Vega, the index for indata supports streaming data. When a data point is added, we increment the count for the value, and decrement it when it is removed. If a data point is modified, we decrement the old value and increment the new value.

The field name argument does not need to be a literal. If the field changes, we build new indexes at runtime. However, if it is a literal, we can build the index on the field while we are generating the code. This ensures there is no lag spike if the index would have initially been built in response to user interaction. Creating an index is a linear time operation, and it adds a constant amount of time to each streaming data operation to keep the index up-to-date, but it speeds up each indata call from linear time to constant time.
3.1.2 The `scale` and `iscale` functions

Another feature which was not supported in the expression syntax is the ability to apply arbitrary scale transformations and inversions. In a predicate definition, any argument can be transformed by a scale or scale inversion, but expressions did not provide this functionality.

The functions `scale` and `iscale` were added to support this functionality. `scale` takes the scale name, the value to transform, and optionally a group on which to lookup the scale. It transforms the value using the scaling function. `iscale` is the same, but performs the inverse scale transform. There is no change to the dataflow graph when using a scale transform in an expression. No dependencies are defined. This is to avoid creating circular dependencies. This also means there is no requirement for the scale name to be a string literal.

3.2 Expression Syntax Extensions

One example visualization that I created for the Vega project was an implementation of a chart from DimpVis [1]. The chart is a scatter-plot of two-variables which change over time. On hover, each point shows a trail of its position over time. The point acts as a scrubber — drag it along its historical path, and the time shown on the graph will change. To implement this interaction, I used vector math to discover whether the point is being dragged towards the previous or next sample point, then I projected the drag location onto an interpolated time somewhere between the two points.

Performing vector math with only scalar operators can be tedious and error-prone. The DimpVis example spec defines almost 20 signal values. The longest definition is for `interYear`, which is the interpolated year value based on the position of the dragged point.

The definition of the signal would be simplified with point and vector operations. In this example, `sqrt(pow(pX-tX, 2) + pow(pY-tY, 2))` is simply the distance between `p` and `t`, and could be handled with a built-in function, with an expression like `dist(p, t)`. I created a set of vector and point functions that would simplify operations like this. However, because all of the functions in the expression language act as globals, this would pollute the global namespace with about a dozen more functions. These can also collide with user-provided signal names. It would be better to add a namespacing feature before introducing so many functions. I was also interested in simplifying vector operations further.

JavaScript does not support operator overloading, but this architecture allows for virtually any rewriting of the expression at compile time, so any expres-
sion syntax that compiles to JavaScript can be supported. As an experiment, I implemented operator overloading for 2D points or vectors. To do this, each overloaded operator was replaced with a custom function. \( \mathbf{a} + 5 \), for example, would become `this.defs.add(a, 5)`. This `add` function would check if either operand was a point, and perform the overloaded addition for points, and otherwise perform the built-in `add` operation. Some limited type-inference eliminated situations where the overloaded call is unnecessary because both arguments were provably not points.

By replacing the parser and AST analysis phase, the expression language could be a completely new language. It could support complex object types and new operators. This flexibility could make it easier to define certain operations, but it places a larger burden on users. Using Vega already requires users to learn many new concepts. Even if the current expression language is imperfect for some tasks, there a benefit in defining it to be a subset of JavaScript, which users will most likely be familiar with. For this reason, these expression syntax extensions were not added to the official releases of Vega.

4 Performance

Because Vega is a declarative language, there is considerable flexibility in its implementation. We can change the internals of the code to improve performance without affecting the user’s usage of it. The largest example of this is the rendering code. We have two different implementations of the renderer. One uses SVG to render the visualization, and another uses HTML canvas. Rendering with the canvas is more complicated. The canvas renderer parses SVG path strings to draw them manually. It has to perform hit detection for user input events to determine what the user interacted with. However, the canvas renderer also has better performance in general. Users can switch to the canvas renderer with no change to their Vega specifications and get a free speed boost. The performance improvements that I have attempted for Vega are similar. They would not require code changes, but could provide a speed-up.

4.1 Dataflow Optimizations

4.1.1 Automated Sorted Indexes

Using the extra metadata collected by the expression parser, we can have better performance for range selection queries, either for predicate selection or filtering. The simplest case is a filter that is a call to the function `inrange(value, a, b, exclusive)`. This function checks whether the given value is within the range of \((a, b)\) or \((b, a)\), either exclusive or inclusive of the end points. Using a sorted data structure, we can efficiently calculate updates for the results of
the selection operation. In the initial implementation, the bounds $a$ and $b$ must be purely global values — they do not depend on the datum in question. The value must depend only on the datum and no global values.

The implementation of this optimization adds a special case to the streaming filter operator, which handles incoming changesets. Added, removed, and modified points are handled exactly the same as the regular filter operation. However, we can also check the changes to the end points $a$ and $b$. When either or both end points change, we can use a binary search to find the new bounds of the selection, and put the differences in the output changeset. A regular filter implementation is $O(n)$ in the size of the data set when an endpoint changes. This is $O(\log n + m)$, where $m$ is the number of changed points. This will have better performance.

The proof-of-concept implementation used a red-black tree implementation from the bintrees package on npm\(^b\). The expression parser was modified to be re-entrant, so that the `inrange` code generator function could parse the arguments into separate expression objects to track individually. There were a few unimplemented features related to static analysis of the code. As stated above, the `inrange` optimization requires certain arguments to `inrange` to only depend on global state or datum field values. The analysis was not perfect, and could result in incorrect behavior. In particular, function calls may be improperly marked. A function call was marked as global- or datum-only if all of its arguments were one of these. This will fail on functions that have hidden global dependencies or are otherwise impure. These troublesome functions include `random` and `now` (impure functions), and `indata` and `scale` (global dependencies not reflected in their arguments).

The test case for the sorted index was a visualization with a one-dimensional brush for selection (figure 3). A filter removed any data points outside of the brush region. The data was binned and aggregated into a histogram before

---

\(^b\)https://www.npmjs.com/package/bintrees
display, so that rendering would not be a bottleneck.

Qualitatively, the implementation did not improve the performance of Vega on the test specification. The frame rate was roughly equal between the two. I did not realize at the time, but the Vega system as a whole does not take advantage of the optimization. Though the filter transform itself may be faster asymptotically, changing signal values cause the changeset that the filter expression receives to contain every data point in its modified list. Constructing this list takes linear time. This means that at best the optimized filter transform can only have better constants, not asymptotic speedup. The red-black tree implementation might give asymptotically better performance for range queries, but even at 10,000 points, its larger constant factors negate any possible speedup.

Though the filter optimizations did not work with the current version of Vega, a new Vega dataflow system could take advantage of the possibilities for optimization. So far, I have reimplemented Vega’s expression compilation from scratch with fine-grained dependency tracking and the ability to mark impure subexpressions. It also makes writing new analysis phases easier. This can be used to let many other operators and operations be optimized. A good place to start would be the cross transform. This transform calculates the cross product of two data sources, with an optional filter expression. The current implementation calculates the full cross product then filters, but a cross product with a filter is a data join operation, like in any database management system. There are many existing techniques for optimizing joins. Vega could use these techniques to efficiently perform the cross transform.

4.1.2 prev_update Optimization

After my attempts to optimize the dataflow graph were unsuccessful, I started running profiles on all of the example Vega specifications. One function that showed a large cost in some examples was prev_update (figure 4). Some streaming data transform operators need to know the previous value of a data point to update when a data point changes. The data source indexes for indata are an example. When a point is modified, it decrements the count for the old value of the field before incrementing the count for the new value. prev_update runs after each cycle of changes in the dataflow graph and keeps the _prev property in sync.

Some of the cost for this function is unavoidable, but not all data sources need to have a _prev field. Streaming data transforms request it for each data source that needs it. I noticed, however, that the implementation did not completely skip the data sources that didn’t need updating. It still checked every data point, so it took linear time with the total number of loaded data points rather than the number that needed to be updated. A quick fix made it skip the data sources that didn’t need a prev_update cycle, reducing the cost.
for many visualizations.

### 4.2 Rendering Optimizations

Another lesson I learned from the profiles that I ran was that rendering is a bottleneck for many visualizations. Because the canvas renderer is the fastest renderer in Vega, I focused my optimization efforts on improving the performance of the canvas renderer.

#### 4.2.1 Minimizing Canvas State Changes

I spent a lot of time on the canvas renderer to try to improve performance. Conventional wisdom to improve canvas performance is to minimize state changes. This means changing the fill color, stroke color, and other properties as few times as possible. This allows the browser to batch draw calls to the GPU. I ran benchmarks which suggested that performance is decreased even if the canvas state is set to the same value. The color is set with an HTML color string such as ’red’, and is converted internally to a RGBA tuple. Setting it to red again will degrade performance just as if it were set to a different color. Microbenchmarks of canvas rendering seemed to confirm this.

Vega renders marks in the order they appear in the specification. Data points are rendered in the order they appear in the data source. Additionally, marks may be semi-transparent, and can include fill colors and stroke colors. This greatly limits the options for optimizing rendering. The goal of my optimization efforts was to be seamless. I couldn’t change the result of rendering. However, I could eliminate the penalty for unnecessary sets of the canvas state. I added a wrapper around the canvas state which ignored any state sets which did not result in a real state change. With this, even a visualization with several colors could decrease state changes by sorting by color.

<table>
<thead>
<tr>
<th>Self</th>
<th>Total</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>24563.5 ms</td>
<td>24563.5 ms</td>
<td>(idle)</td>
</tr>
<tr>
<td>2685.6 ms</td>
<td>2685.6 ms</td>
<td>▲ module.exports.prev_update</td>
</tr>
<tr>
<td>2685.6 ms</td>
<td>2685.6 ms</td>
<td>▲ prototype.synchronize</td>
</tr>
<tr>
<td>2460.3 ms</td>
<td>5169.1 ms</td>
<td>▲ prototype.synchronize</td>
</tr>
<tr>
<td>1246.1 ms</td>
<td>1246.1 ms</td>
<td>▲ program</td>
</tr>
<tr>
<td>822.4 ms</td>
<td>822.4 ms</td>
<td>▲ (anonymous function)</td>
</tr>
<tr>
<td>654.2 ms</td>
<td>965.5 ms</td>
<td>▲ update</td>
</tr>
<tr>
<td>595.2 ms</td>
<td>1331.3 ms</td>
<td>▲ prototype.transform</td>
</tr>
<tr>
<td>590.9 ms</td>
<td>839.9 ms</td>
<td>▲ module.exports.idFilter</td>
</tr>
<tr>
<td>369.1 ms</td>
<td>1142.4 ms</td>
<td>▲ proto_mod</td>
</tr>
<tr>
<td>353.9 ms</td>
<td>353.9 ms</td>
<td>▲ clearRect</td>
</tr>
<tr>
<td>259.2 ms</td>
<td>1461.3 ms</td>
<td>▲ mod</td>
</tr>
</tbody>
</table>

Figure 4: Profile showing the cost of \texttt{prev\_update}
4.2.2 Combining Canvas Paths

![Figure 5: Rendering differences when combining intersecting paths. Separate paths left, combined paths right.](image)

Another piece of conventional wisdom is to combine multiple objects into a single path and fill and stroke them in a single draw call. The symbol renderer in particular creates a new path for each symbol mark instance. Combining paths is not trivial. The result of drawing multiple paths at once is not the same as drawing them individually (figure 5). In the case of drawing overlapping circles, for example, drawing one after the other would make the stroke of one cover the other. The one on top would be the one that is later in the data source or specification. If they are drawn together, both strokes will overlap. Additionally, semi-transparency works differently. If both are drawn separately, the intersecting portion will be darker. If drawn together, the intersecting portion will be the same as the rest.

It is still possible to have the same drawing semantics with combined paths. If all mark instances use a the same opaque color and no have stroke, then changing drawing order and combining paths will not change the result. Otherwise, we could treat the collection of objects to draw as a directed acyclic graph. An object has a dependency on another object if it overlaps the object and is on top of it. The renderer could iteratively draw all objects with no dependencies and the same colors until everything is drawn.

Before attempting to implement this feature, I benchmarked a proof-of-concept of the best case scenario to see if it would be worthwhile. This symbol renderer sets the state once and draws everything in a single path, as it might if there were no overlap. This gives an incorrect render, because all symbols render with the same color and overlap behaves differently. However, it represents the limit of the speed-up combining paths could provide.
4.2.3 Evaluation

I tested three versions of Vega with the vega-benchmarks project. This uses ChromeDriver to automate load the page and perform a fixed set of interactions. The visualization it loads renders a scatter-plot of points. While the mouse is up, the points are semi-transparent gray. While the mouse is down, the points are semi-transparent red. This forces a re-render each time the mouse state changes.

All tests were performed using Chrome 50 on an Ubuntu 15.10 machine with 8 gigabytes of RAM. The three versions of Vega were **vega-stable**, which is an unmodified copy of Vega 2.5.2, **vega-wraper**, a patch on Vega 2.5.2 which wraps the canvas context to reduce canvas state changes, and **vega-combined**, a patch on Vega 2.5.2 which renders all symbols as a single path with a single call to fill and stroke.

The results show that proxying the canvas context to remove state changes had no significant effect — vega-wraper and vega-stable have similar performance. Combining paths was a bit faster. At 10,000 points, it approaches nearly twice the speed. Remember, though, that the combined path implementation was the absolute best case. Vega-stable is rendering 10,000 points separately and calling fill and stroke 10,000 times each. Vega-combined draws all 10,000 points as one path, and makes a single call each for fill and stroke. A path combining implementation that renders correctly would only be able to combine a few symbols at a time — they must be non-overlapping, have the exact same fill and stroke style, and all the points underneath them have to have been rendered already. The cost of discovering this rendering order is not worth the overhead if the speedup is not substantial. Neither optimization seems worthwhile.
Figure 7: Time taken for each mouse state change re-render. Error bars show ±1 standard deviation.

Figure 8: Frames per second vs number of data points.

5 Future Work

My work on Vega has improved Vega as a programming language, but also highlighted much work to be done. The expression parser could perform more complex static analysis of expressions, and operators could use that to optimize their operations rather than treating the expressions as black-boxes. As shown by the failure of the sorted index optimization, this will require extensive changes to the way Vega processes changesets so that the system as a whole can take advantage of asymptotic speed-ups.

The expression language could add namespaces or includes for function extensions. Vector and point operation functions could be added as an optional add-on, so that the global namespace is not polluted by more and more functions.

According to my profiles, Vega’s code is very flat. The time taken is spread
out among many relatively quick functions. Low-hanging fruit like the slow `prev_update` function is fixed. The largest portion of the time is spent in rendering. The canvas renderer also has little room for easy optimization. Further performance improvements will probably have to re-architect the code rather than focus on individual functions. A WebGL rendering backend could also probably improve rendering performance.

References

