Vega Postgres

Andrew Fichman
Advisor: Leilani Battle
University of Maryland
December 2019

Abstract

Vega is a visualization grammar along with a JavaScript run-time that generates flexible and dynamic browser-based visualizations. Vega specifications allow designers to declare external data sources, but these sources must be JSON-formatted files that the run-time loads entirely into memory prior to evaluation. This causes performance issues when datasets are on the order of gigabytes or more, such as sluggish user interfaces and browser crashes. To address this limitation, we provide an extension, Vega Postgres, that offloads storage and computation for Vega's aggregate transforms to a Postgres database server. The extension has two components: a Postgres transform for Vega, along with a middleware server to forward queries from the browser to a database, and an algorithm that rewrites aggregate transform nodes as Postgres transform nodes at the Vega dataflow level. With this extension, we hope to enhance the Vega ecosystem to enable greater scalability.

1 Introduction

Interactive data visualizations are important because they promote data exploration, insight discovery, and hypothesis generation. Vega is an effective and highly flexible interactive visualization system that provides a fully declarative language for specifying visualizations [12]. Vega specification are JSON-formatted files that adhere to a specialized grammar, defining data sources, user-interactions, data transformations, visual encodings, and data dependencies. The Vega compiler parses these specifications into a dataflow graph, which the Vega run-time then executes to generate SVG or Canvas visualizations in a manner that achieves superior interactive performance [17]. The Vega project is browser-based and its functionality is exposed as a set of Node.js modules.

As datasets continue to grow, tools such as Vega need the ability to scale. In its typical usage, Vega's entire procedure of generating visualizations—including loading all data—happens in the browser. This presents two immediate limitations. First, the amount of data that can be visualized using Vega is limited by the amount of data that can fit into memory in the browser. In practice, this limitation restricts dataset sizes to hundreds of thousands or low millions of records, depending on the number of attributes involved. Second, all the computational load involved in performing aggregate transformations on input data (e.g. averages, counts) is placed on the browser application running on the end-user's computer. It is possible to run Vega in a Node.js process, but the Vega dataflow is not as efficient as a relational database management system (RDBMS). Further, data sources would still have to be fully loaded into the Node.js process' memory. Our work seeks to provide the Vega ecosystem with an enhancement to overcome these challenges by offloading storage and computation to the Postgres RDBMS.

1.1 Vega Example

Vega specifications are schematized JSON objects that define interactive data visualizations in a declarative fashion. In Fig. 1 we provide an example Vega specification that defines a simple bar chart. In this section we explain this example, detailing the major properties declared within the specification.

```
{
  "$schema": "https://vega.github.io/schema/vega/v5.json",
  "width": 400, "height": 200, "padding": 5,
  "data": [
    {
      "name": "table",
      "values": [
        {"category": "A", "amount": 28},
        {"category": "B", "amount": 55},
        {"category": "C", "amount": 43}
      ]
    }
  ],
  "scales": [
    {
      "name": "xscale", "type": "band", "range": "width",
      "domain": {"data": "table", "field": "category"},
      "padding": 0.05
    },
    {
      "name": "yscale", "range": "height",
      "domain": {"data": "table", "field": "amount"}
    }
  ],
  "axes": [
    { "orient": "bottom", "scale": "xscale" },
    { "orient": "left", "scale": "yscale" }
  ],
  "marks": [
    {
      "type": "rect", "from": {"data":"table"},
      "encode": {
        "enter": {
          "x": {"scale": "xscale", "field": "category"},
          "width": {"scale": "xscale", "band": 1},
          "y": {"scale": "yscale", "field": "amount"},
          "y2": {"scale": "yscale", "value": 0}
        }
      }
    }
  ]
}
```

Figure 1: Vega specification for a bar chart.

- `$schema`: URL to the JSON schema that the Vega specification uses.
- `width`: pixel width of the visualization.
- `height`: pixel height of the visualization.
- `data`: list containing a single inline data source named `table` whose entries are `(category, amount)` tuples. Note that URLs to JSON-files are also allowed.
- `scales`: defines two scales `xscale` and `yscale`, whose domains are determined by the `category` and `amount` fields, respectively, from the `table` data source.
- `axes`: locations and scales for two axes. One axis uses `xscale` and is located at the bottom of the visualization; another axis uses `yscale` and is located on the left of the visualization.
- `marks`: data encodings for the visualization.
  - `marks.type`: specifies that each mark is rectangle (the visualization is a bar chart).
  - `marks.from`: specifies that the data source for the rectangle marks is `table`.
  - `marks.encode`: specifies the horizontal location (`x`), width (`width`), vertical starting location (`y2`), and height (`y`) of each rectangle mark. Note that these properties are within the `enter` property, indicating that they should be set on initialization.

In Fig. 2 is the visualization generated by Vega from the specification in Fig. 1. Notice that there are three generated rectangle marks that encode the three data values from `table`: `(A, 28)`, `(B, 55)`, and `(C, 43)`.

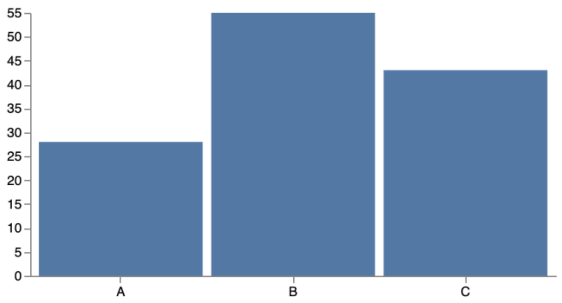

Figure 2: Bar chart generated from the Vega specification in Fig. 1.

1.2 Our Contribution

The above specification provides an example of a dataset that Vega easily handles. Although Vega's capabilities extend further, i.e. to datasets with hundreds of thousands or low millions of tuples, modern datasets often exceed the bounds of these capabilities. The goal of this work is to address Vega's scalability limitations by providing an extension to Vega that offloads storage and aggregate transform computation to a Postgres database server [10]. Our extension has the following two components.

Postgres Transform. This is a new Vega transform (see Section 3.1) that executes a specified query against a Postgres database server during the run-time evaluation of the dataflow graph. The transform—running in the browser—is supported by a lightweight middleware server that forwards its requests to a Postgres back-end. This contribution grants Vega's dataflow the ability to leverage the performance capabilities of an RDBMS, allowing users to specify aggregate queries against large datasets that cannot be handled in the memory of a single browser or Node.js process.

Rewrite Algorithm. Our algorithm (see Section 3.2) rewrites the Vega dataflow graph just before its initial run-time evaluation. This rewrite identifies each top-level Vega aggregate transform node that is downstream from a Postgres data source, replacing it with a Postgres transform node that is configured with an equivalent generated SQL query. Once rewritten, Vega's dataflow is poised to wield Postgres' strength as an RDBMS to support large-scale aggregate queries. The algorithm we provide allows Vega users to immediately reap the performance benefits of an RDBMS back-end: automated query generation means they get scalability benefits for free, with virtually no changes to their Vega specifications.

2 Related Work

There is other work that seeks to improve the scalability of Vega [12]. The `scalable-vega` [14] project provides a custom Vega transform that executes a hand-written, specification-declared SQL query against an OmniSciDB [9] database; this was the primary inspiration for our research. We sought to continue this work by generating SQL queries from the Vega specification, rather than relying on hand-written queries. VoyagerDB [13] is an attempt to provide scalability to Voyager [18] by generating SQL queries directly from Vega-Lite [16] specifications. Here we take a similar approach, but generate SQL queries from the compiler-generated Vega dataflow, rather than a Vega-Lite or Vega specification directly. Finally, `ibis-vega-transform` [5] interprets transforms from Vega specifications as Ibis [4] expressions, which are then executed against a SQL database. Again, `ibis-vega-transform` operates on Vega specifications directly, whereas we operate on Vega dataflows.

Outside the Vega ecosystem there are efforts, e.g. SQLite [11] and DuckDB [15], that embed database systems within a process; there are also APIs for connecting remote databases to code, such as ODBC [6] and JDBC [3]. However, it can be difficult to apply these tools to specific application contexts, including interactive data visualization, without spending considerable resources developing an ad hoc solution. What we provide with Vega Postgres is a system-level solution for database connectivity that obviates the need for Vega users to write their own database client code. Instead, they need only declare Postgres transforms to source relational data in their Vega specifications (see 3.1). Our algorithm then automatically generates database queries from these transforms (see 3.2). The advantage of this approach is that it preserves Vega's declarative interface, hiding the complexity of targeted query generation and database communication at the implementation level. This minimizes the burden on Vega users since they only have to learn the simple interface for one new transform.

3 Extension Components

To improve the scalability of Vega, we provide an extension that offloads data source storage and computation for Vega aggregate transforms to a Postgres database server. This extension has two components. The first component is a new Vega transform that executes a specified query against a Postgres database server during the run-time evaluation of the dataflow graph. Code for this component can be found in the `vega-transform-pg` Node.js module here. This component is supported by a lightweight middleware server that forwards requests from the browser to a Postgres database server. The second component is an algorithm that generates SQL queries from top-level aggregate transforms that are downstream from those data sources; the algorithm replaces each such aggregate transform with a VTP containing an SQL query with matching semantics. Code for the middleware server and the second component—along with example Vega Postgres specifications and a README explaining demo setup—can be found here. In the next section, we discuss the first component in more detail.

3.1 Postgres Transform

Here we describe the first component of our Vega extension. It is comprised of two parts: a custom Vega transform that creates Postgres query requests and a middleware server that forwards these requests from the browser to a Postgres database server. We describe the transform's implementation in section 3.1.1 and the middleware server in section 3.1.2.

3.1.1 Transform Implementation

The Vega language is highly extensible. One way in which developers can extend it is by adding custom transforms. This involves extending the base `Transform` class from Vega's Node.js `transform` module [12].

```
"data": [{
  "name": "cars",
  "transform": [{
    "type": "postgres",
    "relation": "cars"
  },
  {
    "type": "aggregate",
    "fields": ["miles_per_gallon"],
    "ops": ["average"],
    "as": ["miles_per_gallon"],
    "groupby": ["cylinders"]
  }]
}],
"marks": [{
  "type": "rect",
  "from": {"data": "cars"},
  "encode": {
    "enter": {
      "x": {"scale": "xscale", "field": "cylinders"},
      "width": {"scale": "xscale", "band": 1},
      "y": {"scale": "yscale", "field": "miles_per_gallon"},
      "y2": {"scale": "yscale", "value": 0}
    },
```

Figure 3: Snippet from a Vega specification with a Postgres data source.

The primary function that derived types override to achieve custom functionality is the `Transform.transform` function, which is invoked at run-time as the Vega dataflow is executed, either due to change propagation triggered by user interaction or on initialization [17]. The `Transform.transform` function takes data tuples as input, performs some computational transform on them, and then generates new data tuples as output. We have developed a new Node.js module `vega-transform-pg` that exports `VegaTransformPostgres` (VTP), a custom transform type that is responsible for executing SQL queries against a Postgres database server. VTPs act as data sources via placement in the transform array for an entry in the data property for a Vega specification.

In Fig. 3 is an example that uses a VTP to declare a Postgres datasource. The portion of the specification that is of interest is the array entry named `cars` in the top-level `data` property. The `cars` entry contains an array of transforms in its `transform` property, the first entry of which declares a VTP by setting its `type` property to the value `postgres`; the `relation` property of this transform indicates that the relation or table from which this VTP will draw data is named `cars`. At run-time, the downstream transform (i.e. the next entry in the `transform` array) of type `aggregate` will source its data tuples from the Postgres relation `cars`.

Generally, each VTP is configured with the following:

1. HTTP server options
2. Postgres connection string
3. Relation name
4. Generated SQL query

Since VTPs run in the browser, they do not directly communicate with a Postgres database server. Instead, they forward queries through a lightweight middleware server (see Section 3.1.2) specified by configuration (1) (see Fig. 4). Configuration (2) determines which Postgres database server to query and is of the form `postgres://host:port/database`. Both configurations (1) and (2) are specified programmatically via the `setHttpOptions()` and `setPostgresConnectionString()` functions, respectively. Configuration (3) is a VTP parameter that users declare in their Vega specifications to determine the Postgres database relation from which data tuples are gathered. Finally, configuration (4) is the actual SQL query, which is generated just before the execution of the Vega dataflow (see Section 3.2).

```
const httpOptions = {
  hostname: 'localhost',
  port: 3000,
  method: 'POST',
  path: '/query',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  }
}
```

Figure 4: HTTP options for a VTP instance.

VTPs are atypical Vega transforms in that they do not take any input data tuples from the Vega dataflow; rather, the input data tuples can be thought of as the tuples from the Postgres database relation the VTP targets. When a VTP's transform function is executed as part of dataflow graph execution in the browser, it sends an HTTP request containing an SQL query to a middleware server and waits for the results. The middleware server forwards the query to a Postgres database server and then returns the query results to the browser. When the VTP in the browser receives these results, it emits them for propagation to downstream nodes in the dataflow graph. It is important to note that VTP transform functions are blocking, synchronous operations.

3.1.2 Middleware Server

When a VTP's transform function is invoked it sends an HTTP request containing an SQL request

to a middleware server that we developed as part of this project. The middleware service is a lightweight Node.js Express server [1] that forwards SQL query requests from a browser client to a Postgres database. The server exposes two routes, `/createSql` and `/query`. The route `/createSql` is a utility route used during development and testing that creates and populates a Postgres relation from a list of JSON tuples. This route expects the HTTP request body fields `postgresConnectionString` (of the form `postgres://host:port/database`), `data` (a list of tuples to insert into the relation), and `name` (the name of the relation to create). The `/query` route forwards an SQL query to a Postgres database is the route of primary importance, as it is the route requested by a VTP's transform function. This route expects the HTTP request body fields `postgresConnectionString` (again, of the form `postgres://host:port/database`) and `query` (the SQL query to execute). Both `/createSql` and `/query` are synchronous routes that only return responses to the browser client when their computations are complete.

The middleware server limits the number of Postgres connections by communicating with the Postgres database server using `pg.Pool` from the `node-postgres` Node.js module [7], and it inserts tuples efficiently for the `/createSql` route by using the `pg-format` Node.js module [2].

3.2 Rewrite Algorithm

The Vega compiler parses specifications into a Vega dataflow, which is a directed acyclic graph with nodes that contain information about data sources, transformations, user interactions, and visual encodings. One specific type of transformation node is the aggregate transform node, which summarizes a stream of input tuples to generate stream of output tuples. Each aggregate transform specifies input fields, aggregate operations (e.g. count, average, standard deviation), groupings, and output field aliases. Our algorithm rewrites these aggregate transform nodes as VTPs with generated queries when they are identified to be top-level nodes that are downstream from a specification-declared VTP. In section 3.2.1 we provide some diagrams that show how the dataflow graph is altered, and in section 3.2.2 we cover the details of the rewrite algorithm.

3.2.1 Rewrite Examples

The algorithm we provide identifies each top-level Postgres-dependent aggregate node in the dataflow (i.e. nodes of type `aggregate` that are downstream from a specification-declared VTP with no intervening `aggregate` nodes) and then overwrites its transform function with a VTP transform function that is configured with an SQL whose semantics match those of the original aggregate operation. See Fig. 5 for a simplified example of how the dataflow is altered and Fig. 7 for a real-life example generated from the Vega specification provided in Fig. 6.

In Fig. 5 we see the top-level aggregate nodes (outlined in red) downstream from the specification-declared VTP (outlined in purple, before) are rewritten as VTP nodes (outlined in purple, after) containing SQL queries with corresponding semantics.

In Fig. 7 we see a rewrite of a full-detail dataflow generated from the Vega specification in Fig. 6. Node 10 corresponds to the `postgres` transform contained in the `cars_postgres` data entry from the specification. Node 13 corresponds to the `aggregate` transform contained in the `cars_avg` data entry, which uses `cars_postgres` as its datasource. In the rewrite, node 13 is rewritten as a VTP configured with a semantically equivalent SQL query and node 10 is removed.

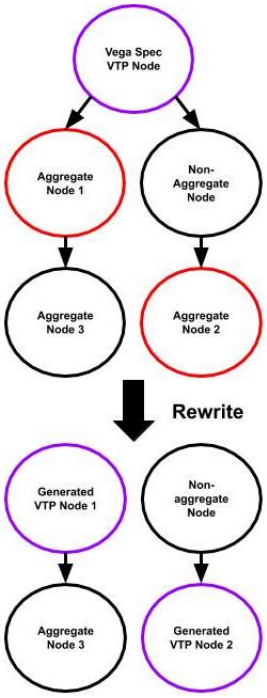

Figure 5: Simplified dataflow rewrite example.

```
{
  "$schema": "https:\/\/vega.github.io\/schema\/vega\/v5.json",
  "width": 400, "height": 200, "padding": 10,
  "data": [
    {
      "name": "cars_postgres",
      "transform": [{
        "type": "postgres",
        "relation": "cars"
      }]
    },
    {
      "name": "cars_avg",
      "source": "cars_postgres",
      "transform": [{
        "type": "aggregate",
        "fields": ["miles_per_gallon"],
        "ops": ["average"],
        "as": ["mpg_avg"],
        "groupby": ["cylinders"]
      }]
    }
  ]
}
```

Figure 6: Simple Vega specification with a VTP and one downstream aggregate transform.

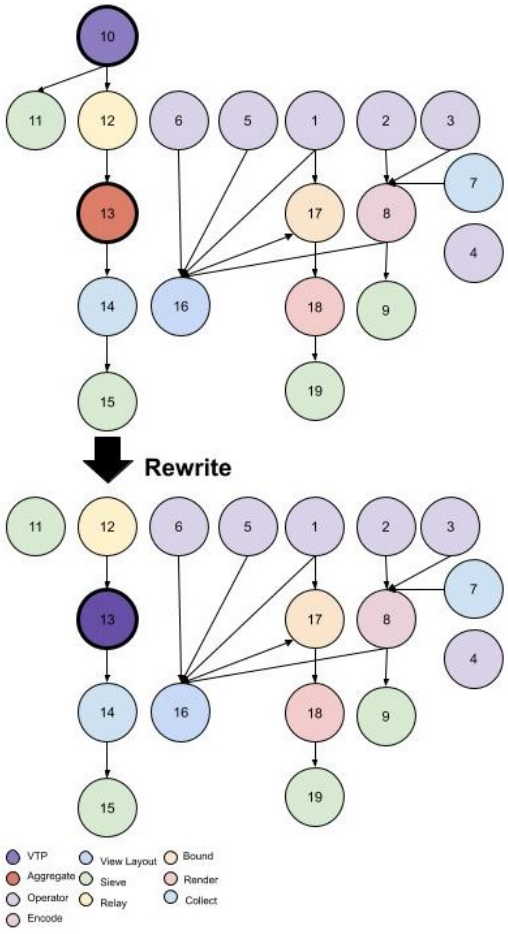

Figure 7: Full-detail dataflow rewrite example.

3.2.2 Algorithm Details

Here we cover the details of the rewrite algorithm. Users writing a Vega specification can declare a VTP as a datasource by adding it to the `transforms` array of an entry in the specification's `data` property (see Fig. 6). When the Vega compiler translates the Vega specification to a dataflow graph, a node will be generated for each specification-declared VTP. Our algorithm then traverses the dataflow graph and generates a replacement VTP for each top-level aggregate transform node that is downstream from a specification-declared VTP node. Each generated VTP contains an SQL query that is semantically equivalent to the original Vega aggregate transform node that it replaces. The algorithm, which rewrites the dataflow graph after compilation but before execution, is composed of the following functions in pseudocode:

```
1  // Returns a percentile Postgres function
2  // for the given field and fraction.
3  function percentileContSql(f, frac):
4    return `PERCENTILE_CONT(${frac})
5      WITHIN GROUP (ORDER BY ${f})`
6
7  // Maps a Vega operation to a Postgres
8  // function.
9  function vegaOpToSql(op, f):
10   switch op:
11     case "average":
12       return `AVG(${f})`
13     case "count":
14     case "valid":
15     case "missing":
16       return `COUNT(*)`
17     case "distinct":
18       return `COUNT(DISTINCT ${f})`
19     case "sum":
20       return `SUM(${f})`
21     case "variance":
22       return `VARIANCE(${f})`
23     case "variancep":
24       return `VAR_POP(${f})`
25     case "stdev":
26       return `STDDEV(${f})`
27     case "stdevp":
28       return `STDDEV_POP(${f})`
29     case "stderr":
30       return `STDDEV(${f})/SQRT(COUNT(${f}))`
31     case "median":
32       return percentileContSql(f, 0.5)
33     case "q1":
34       return percentileContSql(f, 0.25)
35     case "q3":
36       return percentileContSql(f, 0.75)
37     case "min":
```

```
      return `MIN(${f})`
    case "max":
      return `MAX(${f})`
    default:
      throw Error(
        `Unsupported op: ${op}`)

// Generates a semantically equivalent
// SQL query from a Vega aggregate node.
function genAggregateQuery(node, relation):
  let q = SQLQuery()

  // Generates SELECT clause
  for f in node.fields:
    let attr = SQLAttr()
    if f in node.ops:
      attr = vegaOpToSql(node.ops[f], f)
    else
      attr = f
    if f in node.as:
      attr.alias = node.as[f]
    q.select = union(q.select, {attr})
  q.select = union(q.select, node.groupby)

  // Generates GROUPBY clause
  q.groupby = node.groupby

  // Generates FROM clause
  q.from = relation

  // Generates WHERE clause
  for f in node.ops:
    if node.ops[f] == "missing"
      let cond = SQLCond()
      cond.attr = f
      cond.cond = "IS NULL"
    if node.ops[f] == "valid":
      let cond = SQLCond()
      cond.attr = f
      cond.cond = "IS NOT NULL"

// Overwrites the given node's transform
// function with a VTP transform function,
// and then recurses on its targets.
function rewriteTopLevelAggregates(node, vtp):
  if isAggregate(node):
    let query = genAggregateQuery(
      node, vtp.relation)
    node.transform = VTP(query)
    return
  for t in node.targets:
    rewriteTopLevelAggregates(t, vtp)

// Generates a SELECT query for fields.
function genFieldsQuery(fields, relation):
  let query = SQLQuery()
  query.select = fields
  query.from = relation
  return query

// Returns non-aggregate fields for node.
function nonAggregateFields(node, vtp):
  if isAggregate(node):
    return {}
  let f = node.fields
  for t in node.targets:
    f = union(
      f, nonAggregateFields(t, vtp))
  return f

// Generates aggregate and non-aggregate
// queries for the given node.
function genQueries(node):
  rewriteTopLevelAggregates(node, node)
  let naf = nonAggregateFields(node, node)
  if naf != {}:
    node.query = genFieldsQuery(
      naf, node.relation)

// Handles each specification-declared
// VTP by rewriting its top-level
// dependent aggregate nodes with VTP
// functions configured with semantically
// equivalent SQL queries.
function rewrite(dataflow):
  for node in dataflow:
    if isPgTransform(node):
      genQueries(node)
      if !node.query:
        remove(dataflow, node)
```

Listing 1: Algorithm to rewrite aggregate transforms.

The SQL query generation is straightforward. Aggregate transform nodes in the dataflow graph are JavaScript objects that contain properties for input fields (`fields`), aggregate operations (`ops`), groupings (`groupby`), and output field aliases (`as`). These map almost directly to SQL. Consider the following aggregate transform:

```
{
    "type": "aggregate",
    "fields": ["f1", "f2", "f3"],
    "ops": ["average", "stdev"],
    "groupby": ["f3"],
    "as": ["a", "sd"]
}
```

Listing 2: Example aggregate transform.

From such an aggregate transform, our algorithm would generate the following SQL query:

```
SELECT AVG(f1) AS a, STDDEV(f2) AS sd, f3
FROM R
GROUP BY f3
```

Listing 3: SQL generated from aggregate transform.

Vega aggregate operators do not map exactly to Postgres aggregate functions, but they are close. There are two Vega aggregate operators that result in a `WHERE` clause in the generated SQL query, namely `valid` and `missing`. The `valid` operator applied to a field `f` will generate `WHERE f IS NOT NULL`; the `missing` operator applied to a field `g` will generate `WHERE g IS NULL`. Multiple `valid` or `missing` operators are supported, and result in `IS NOT` and `IS NOT NULL` conditions chained with `AND` logical operators. As of now, all Vega aggregate operators are handled except `argmax`, `argmin`, `ci0`, and `ci1`. There is one special case to mention, namely where there are no aggregate transform nodes downstream from a given specification-declared VTP. In this case the specification-declared VTP must remain in the dataflow graph along with a generated query that selects all fields from its specified relation that are mentioned in downstream encode and extent nodes, since these are used to generate marks in the final visualization.

4 Technical Details

In this section we describe the Vega Postgres architecture, the contents of important files, and how to get an example system up and running.

4.1 Architecture

Here we detail Vega Postgres' architecture, which is composed of the following applications and processes.

Browser. The browser is what actually runs Vega JavaScript code—augmented with our Vega Postgres rewrite algorithm and VegaTransformPostgres (VTP) custom transform—and generates HTTP requests. We have provided a demo (see 4.3) browser-based application that exposes a simple user interface for experimenting with Vega Postgres. The user can do either of two tasks: (1) upload JSON data to populate a relation in Postgres, or (2) upload a Vega JSON specification to generate a visualization. The path in (2) is what is outlined in Fig. 8.

Webserver. The webserver (excluded from Fig. 8 for brevity) simply serves up index files that run in the browser.

Middleware server. The middleware server, described in 3.1.2 in detail, is a Node.js Express server that accepts from the browser HTTP requests containing SQL queries, forwards the SQL queries to Postgres, and then returns the resulting tuples back to the browser.

Postgres. Postgres is the RDBMS that houses and provides SQL query access to the relational data.

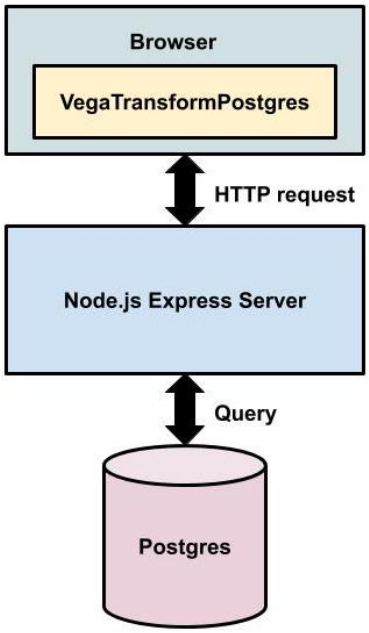

Figure 8: Vega Postgres architecture.

4.2 Important Files

The Vega Postgres extension is a JavaScript project whose dependencies are managed with npm [8]. The files for the Vega Postgres extension can be found can be found on github in two repositories. The first repository `scalable-vega` contains code for the demo client, middleware server (see 3.1.2), and rewrite algorithm (see 3.2). The second repository `vega-transform-pg` contains code for the VTP (see 3.1.1) and is published as an npm package here that is listed as a dependency for `scalable-vega`. Here is an overview of the directory structure and important files of each repository.

The `scalable-vega` repository:

- `data`: directory containing example JSON files that can be used to populate a Postgres database through the demo application.
- `lib/dataflow-rewrite-pg.ts`: contains the dataflow rewrite algorithm's code. Exports `dataflowRewritePostgres()`.
- `index.html`: embeds `main.ts`.
- `main.ts`: demo application that runs in-browser. Exposes a UI with two buttons, one to upload

JSON data to populate Postgres and another to upload a Vega specification.
- `server.js`: contains the middleware server code. This server exposes an API with the routes `/createSql` and `/query` for loading data into Postgres and querying it, respectively.
- `specs`: directory containing example Vega JSON specifications that are supported by Vega Postgres.

The `vega-transform-pg` repository:

- `index.js`: exports `VegaTransformPostgres` (VTP) which is a Vega transform implementation type that communicates with the middleware server to execute SQL queries during Vega dataflow execution at run-time.

4.3 Installing and Running the Code

Perform the following steps to install the demo and its dependencies:

1. Install git.
2. Install Python 2.7 (required by yarn).
3. Install and start Postgres.
4. Create a Postgres db named `scalable_vega`.
5. Run `cd /path/to/dev/repos`.
6. Run `git clone` scalable-vega.
7. Run `cd scalable-vega`.
8. Run `npm install` to install npm dependencies.

Once the demo has been installed, run it with the following steps:

1. Run `yarn start` to start the web server.
2. In another terminal window, run `cd /path/to/dev/repos/scalable-vega`.
3. Run `yarn start:server` to start the middleware server.
4. Open a browser tab to `localhost:1234`.
5. Upload the cars dataset from `data/cars.json` under the path to your `scalable-vega` installation.
6. Upload the cars Vega spec from `specs/cars_average_sourced.json` under the path to your `scalable-vega` installation.

5 Limitations

The Postgres extension for Vega is limited to aggregate transforms. One extension would be to broaden its scope to handle other transforms (e.g. filters) to further push storage and computational load out of the browser. Another limitation of our work is that it does not address the case where the results of an aggregated query are themselves large. In this case, the query will be executed against a Postgres database relation and the results returned to the browser will be a subset of all the tuples in that relation, but even this may prove to be too much data for the browser to handle in some cases. One possible solution here would be to detect when such cases occur and then have the middleware server execute sampling queries rather than the full aggregate queries. A final limitation is that the handling of top-level `encode/extent` nodes is limited to the case where they do not reference a combination of fields, some from Postgres database relations and others from in-line or JSON file sources. Proper handling of fields from mixed sources would involve filtering out non-Postgres fields during the `SELECT` query construction (see Section 3.2).

6 Further Work

One important tool built on top of the Vega ecosystem is the Voyager visualization recommendation system [18]. Voyager is a browser-based tool that generates multiple recommended visualizations at once, all of which are specified in Vega. Our Postgres extension to Vega could be applied to the Voyager project to improve its performance through lower CPU and memory usage. The value of our Postgres extension could be validated by plugging it into performance tests of Voyager on large datasets. In addition to performance testing, our extension's scope could be increased by adding support for other Vega transformation types, such as filters. Supporting filter transforms would require the ability to interpret Vega's filter expressions as semantically-equivalent SQL `WHERE` clauses. Finally, the performance of our extension could be enhanced by employing predicted-load-based sampling techniques in the middleware server, rather than always executing full-fidelity Postgres queries.

7 Conclusion

While Vega is a flexible visualization language that allows designers to declare external data sources, these sources must be JSON-formatted files that the run-time loads entirely into memory before evaluation. This causes performance issues at scale where the number of tuples exceeds a

few hundred thousand. In this work, we have provided an extension that addresses this issue by offloading storage and computation for Vega's aggregate transforms to a Postgres database server. This extension, composed of a new specialized Postgres Vega transform, a middleware server, and a just-before-run-time rewrite algorithm, enhances the Vega ecosystem with a way to improve scalability.

References

[1] Express - Node.js web application framework, 2017. https://expressjs.com.

[2] node-pg-format, 2017. https://www.npmjs.com/package/pg-format.

[3] Database JDBC Developer's Guide and Reference, 2019. https://docs.oracle.com/cd/B19306_01/java.102/b14355/toc.htm.

[4] Ibis: Python data analysis framework for Hadoop and SQL engines, 2019. https://github.com/ibis-project/ibis.

[5] ibis-vega-transform: Python evaluation of vega transforms using ibis expressions, 2019. https://github.com/Quansight/ibis-vega-transform.

[6] Microsoft Open Database Connectivity (ODBC), 2019. https://docs.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc.

[7] node-postgres, 2019. https://node-postgres.com.

[8] npm: build amazing things, 2019. https://www.npmjs.com.

[9] OmniSciDB: Open Source Analytical Database & SQL Engine, 2019. https://www.omnisci.com/platform/omniscidb.

[10] PostgreSQL: The World's Most Advanced Open Source Relational Database, 2019. https://www.postgresql.org.

[11] SQLite, 2019. https://www.sqlite.org/index.html.

[12] Vega: A Visualization Grammar, 2019. https://vega.github.io/vega.

[13] Andrew Fichman, Deepthi Raghunandan, and Jessica Thompson. Extending Voyager. https://github.com/heavyairship/voyager, https://github.com/heavyairship/voyager-server, 2018.

[14] Dominik Moritz. Scalable vega: a demo of scaling vega to billions of records, 2019. https://github.com/vega/scalable-vega.

[15] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1981–1984, New York, NY, USA, 2019. ACM.

[16] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, Jan 2017.

[17] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2016. http://idl.cs.washington.edu/papers/reactive-vega-architecture.

[18] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, Jan 2016.