# Jupyter on Steroids

## ABSTRACT

Despite the plethora of recently proposed tools and frameworks, common data analysis tasks such as data retrieval, curation and visualization remain laborious and non-trivial. The limited flexibility of existing frameworks, pushes code-literate data analysts to interactive notebooks such as Jupyter.

In this work, we argue that interactive notebooks are suboptimal for commonly performed tasks, with regards to their ease of use and interactivity. We propose ViDeTTe, a framework designed to facilitate data analysis through the utilization of a template language. We implement a sample data analysis work flow and present the benefits of using ViDeTTe instead of commonly used imperative language in iPython and Jupyter notebooks.

## 1. INTRODUCTION

User-friendly data analysis tools and frameworks often provide limited flexibility, as they usually focus on a predetermined set of use/analysis cases or a small fraction of the typically large data analysis pipeline. This lack of flexibility often pushes code-literate analysts towards the use of interactive notebooks such as Jupyter.

Interactive notebooks allow the use of popular, high-level and highly expressive imperative languages, such as Python, for analyzing data and composing the results into an easily readable notebook-like interface. Due to the wide popularity of such languages, there is also a huge collection of third-party libraries that can be used by data scientists as building blocks of a much bigger analytical process. Furthermore, the web environment of notebooks enables collaboration between data scientists, since it allows them to directly interact with the user interface in order to develop and run code, process data, generate visualizations, and lastly, compose their findings into an interactive (and re-runnable) report-like page, that contains code, visualizations and textual description of the analysis.

However as we show in this work, interactive notebooks are still suboptimal with regard to ease of use and interactivity. Setting up notebook environments and dependencies, obtaining and combining data and generating the respective visualizations, requires technical knowledge that often exceeds the skill-set of a typical data scientist. Lastly, while such notebooks support the generation of interactive visualizations, this interactivity is not an integral part of the data analysis process.

We address these issues, by extending interactive notebooks with a template language called ViDeTTe. The main contributions of this extension are:

- *Declarative semantics:* ViDeTTe implements formal declarative *Model-View-View-Model* (MVVM) semantics.

  *Fill in why this is a good thing. I have no idea.*

- *Expressive template language:* Prior database work, treats a page as a database view. Building on that, our template language goes beyond SQL query and view definition in both style and fundamental expressiveness. It is a mixture of query as well as web templating language that works on ordered (arrays) and semi-ordered (JSON) data.

- We allow in-line declarative code directly in JSON...

In this work, we demonstrate the use of ViDeTTe via a walkthrough example. Specifically, we want to use website access data to plot an access count over time histogram. We also want to plot the recorded user demographics (with focus on age groups). We then want to have the ability to interact with the histogram plot and select a time region. This action should automatically update the second plot with the user demographics in the selected time window.

Without loss of generality, we assume a Jupyter server, where the analysts develop their notebooks and a different database server where data is stored. To retrieve the entirety of the required data, we have to query two different databases and join the returned JSON files. Figure ?? shows how our databases are organized. Our fictional analyst will perform the following high-level tasks:

- Data retrieval from remote databases.
- Data curation: Join data and prepare for visualization.
- Data visualization.

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

The remainder of this paper is organized as follows: Sections 2 – 4 present a direct comparison of using ViDeTTe and an imperative language such as Python in order to complete the tasks of our example. Throughout these sections, we demonstrate some of the main contributions of ViDeTTe. Section 5 provides further discussion regarding our proposed extension and presents other useful aspects of it not used in our walkthrough. Finally, Section 6 concludes the paper.

# 2. DATA RETRIEVAL

*It might be a good idea to merge these 3 sections into 1 called "Walkthrough example" or something like that. Depends on how large these sections become.*

Data retrieval is one aspect where ViDeTTe shines. The analyst is source specific database queries in order to retrieve data from the respective database servers. In a Python implementation, analysts need to perform a series of tasks before being able to retrieve data. Some of these tasks potentially fall way beyond the skill set of the average data analyst.

*Costas: We should also say that we simplify file imports. For cases, when the data analysts wants to use csv or json files from their local computer, they can use an import button, which is part of our UI, to upload them to the notebook server. After that step is completed we trigger a function that automatically converts this file to JSON and we assign it to a ViDeTTe variable, so that it can be used in the notebook. Currently, this task requires ssh-ing or scp-ing the files to the notebook server, and potentially moving files to the directive/file-system that is seen by the notebook, which also requires system administrator-linux knowledge*

First, our analyst needs to install and configure database drivers. This step will either introduce a dependency between the analyst and some system administrator, or the analyst will need to have the required access rights in order to perform the installation. The later can be a security concern or can lead to corrupted systems if not performed correctly. The complexity of this step increases as the number of different database systems, our analyst wants to access, increases. More specifically, for cases when an analyst needs to access data stored in a MongoDB and a MySQL database, two drivers will need to be installed.

Once the system is configured, the analyst then needs to read lengthy documentation documents in order to properly issue queries to the databases via imperative code by using the library API, then consume the results by using internal constructs of the API, and potentially convert them into a format that assists in data processing. Lastly, the analyst will have to convert the format of the data again in order to feed them into the respective visualization library.

Finally, when implemented in a Jupyter notebook, the analyst's credentials for the database server might lie on plain sight to anyone who has access to the notebook, disrupting the valuable ease of results communication through notebooks.

ViDeTTe addresses each one of the aforementioned imperative code issues efficiently: The analyst generates a configuration file, containing information required for establishing a successful connection with the respective database systems. Figure 1a shows an example of the configuration file that is used for connecting to a postgres database which contains the data that will by used in our analysis. The configuration file must contain the type of the database system, the host name, port and the credentials that will be used for obtaining a connection. Additionally, it contains the database tables that will be accessible by queries, only the tables that are explicitly defined in the configuration file will be accessible in the notebook. After this configuration file is imported, (which is done directly from the notebook's UI), it will be hidden from the UI and encrypted so that this information is no longer visible to the notebook reader. Furthermore, as an added benefit, the schema of the accessible tables will be displayed on the UI, thus allowing the notebook user to get a quick glimpse on it, when generating queries.

Once this step is done, the analyst is free to issue queries in order to access database systems. Figure 4a contains a sample query that is used in our analysis. The query joins the two tables: *visitors* and *page_views* on the id of the visitor, then groups the result on the *time* attribute and runs a *count* aggregate to count the visitors per unit of time. Lastly, it sorts the resulting dataset by time in ascending order. Figure 4b shows the result of the query. Notice that the result has been converted into a JSON array by ViDeTTe. In the next section we will show how to use this dataset to generate our first visualization, without the use of any imperative logic.

## 2.1 Source Wrappers

ViDeTTe contains a set of source wrappers that enables data retrieval from various different sources both relational and non-relational. These source wrappers come pre-installed with ViDeTTe so that the notebook user will not have to take on any system administrator duties. The user simply provides the configuration file and writes the query in the language supported by each database system. The source wrapper uses that information to connect to the respective database system, retrieve the requested data, convert it into JSON format and make it available to the user. For database systems that do not contain tables with schema, the notebook user has to provide the respective record container under the "expose" attribute of the configuration file (i.e if the user accesses a MongoDB database, she will have to declare a collections of documents instead of database tables). Additionally, in cases when the accessed database system does not have a schema the notebook will simply display a small portion of the dataset, thus assisting the notebook user to compose queries.

*Costas: Maybe put the last two sentences in a side-note?*

# 3. DATA CURATION

DATA CURATION

# 4. VISUALIZATION

*Costas: Mention that the problems are: 1) Initial installation 2) Data conversions, 3)Use of imperative code, 4) Most of these libraries do not generate interactive visualizations*

The main challenge when generating visualizations in interactive notebooks is the variance between the APIs of the visualization libraries. For each visualization library the analyst decides to use, she has to perform an installation of the respective packages (which requires advanced technical knowledge on its own), read lengthy documentation pages that dictate how to use functions provided by each library and then engage in tedious imperative programming in order to "massage" the existing datasets into a set of formats

```
  sources : [ {
    driver   : "postgres",
    host     : "edu.db.domain",
    expose   : [ {
     schema : 'website_info',
     tables: [visits, page_views] } ]
    port     : 5432,
    username : "dbadmin"
    password : 'myP@ss'
  }]
```

(a) DB Access Configuration file

```
<% let readings =
   SELECT count(time) as visits, time
   FROM (SELECT * FROM page_views pv
               join visitors v
         on pv.v_id = v.vid) AS joined_table
   GROUP BY time
   ORDER BY time ASC %>
```

(b) Data retrieval

```
readings = [
   {visits: 15, time: '08:00'},
   {visits: 10, time: '09:00'},
   {visits: 25, time: '10:00'},  ...]
```

(c) Query Result

```
  <% unit highcharts %> {
    title: 'Visitor information' ,
    xAxis : {
      labels : ['08:00','09:00'...],
      min : '08:00'
      max : '22:00'
    }
    series: [{ data: [ {y:15}, {y:10}...] }]
  } <% end unit %>
```

(d) Unit with evaluated unit state

```
  <% unit highcharts %>
  {
    title: 'Visitor information',
    xAxis : {
      labels : [
        <% for reading in readings %>
          <% print reading.time %>
        <% end for %>],
      min : =<%= min_time,
      max : =<%= max_time
    }
    series: [{
      data: [ <% for reading in readings %>
          {
            y  : <% print reading.count %>
          }
        <% end for %> ]
    }]
  }
  <% end unit %>
```

(e) Template `temp_view`

Figure 1: Template, template instance, and UAS configuration file for the running example

```
<% template temp_view() %>
  <% import functions %>
  <% import actions %>

  <% let readings = select t.temp
                    from db.temperature as t
                    order by timestamp %>

  <% unit highcharts %>
  {
    title: { text: 'Temperature monitor' },
    series: [{
      data: [
        <% for reading in readings %>
          {
            y    : <% print reading.temp %>,
            color: <% print toHex(reading.temp, threshold) %>
          }
        <% end for %>
      ],
      lineWidth: 1
    }],
    <% event onSelection redrawSelected() %>
  }
  <% end unit %>

  <% unit slider %>
  {
    min  : 0,
    max  : 10000,
    value: <% bind threshold = 65 %>
  }
  <% end unit %>
<% end template %>
```

(a) Template `temp_view`

```
<% unit highcharts %>
{
  title: { text: 'Temperature monitor' },
  series: [{
    data: [
        { y: 55, color: '#359435' },
        { y: 57, color: '#359823' },
        { y: 56, color: '#359533' },
        { y: 53, color: '#359220' }
    ],
    lineWidth: 1
  }]
}
<% end unit %>

<% unit slider %>
{
  min : 0,
  max : 10000,
  value: 65
}
<% end unit %>
```

(b) Template instance

```
   sources : [ {
     driver   : "postgres",
     host     : "localhost",
     port     : 5432,
     aliases  : [{db : "sensorDb"}]
     username : "dbadmin"
   }]
```

(c) UAS configuration file

Figure 2: Template, template instance, and UAS configuration file for the running example

accepted by each employed API function.

After the data analyst has completed this tasks, she ends up with visual components that, while they can be informative, they cannot be used for further data exploration, without the need of additional imperative code. Specifically, depending on the employed visualization libraries, the data analyst either ends up with non-interactive visualizations, that are essentially static images, or with visualizations, that while they are interactive, this interactivity cannot be used as a part of the analysis, as it will not trigger any changes to other parts of the notebook. In either case, the analysis does not gain much value from such visual components.

## 4.1 ViDeTTe Visual Units

To shield analysts from the laborious task of constructing visualizations, ViDeTTe abstracts out each visual component as a ViDeTTe *visual unit* (or simply *unit*). In the eyes of the analyst a visual unit is simply a black box that takes as input a JSON value containing the data that describes the visualization, namely, unit instance, and the visual unit is responsible for generating the respective visual component. As such a particular instantiation of the unit can be described as `<% unit` $U$ `%>` $v$ `<% end unit %>`, where $U$ the type of the visual unit and $v$ the JSON value corresponding to the input of the unit.

For instance, Figure 1d shows a unit instance of type `highcharts`. The unit instance describes all the information that will be displayed in the visualization (such as the title of the chart, the labels on the x axis and so on). Each visual unit, comes with a unit instance schema that describes the format of unit instance.

## 4.2 Template

*Templates* are declarative specifications used to generate ViDeTTe variables or construct unit instances. A template specifies this function through a set of *template directives*, so that it provides syntax similar to well-known template languages, while it is essentially an expression of a functional programming language without recursion. A template also allows data collection from the visualizations; such collected data can be used in other parts of the template.

There are four template directives: 1) the bind directive, which is encoded as `<%= E %>`, a for-loop, encoded as `<% for v in E %>`, `<% let E%>`, and `<% bind %>`. These are used to describe computation, define variables and set up data collection. We next describe each of them in detail.

**Defining variables.** A template may define variables that are added to the UAS instance so that they can be used in subsequent computation. Variable definition is facilitated by the `let` directive.

The `<% let` $x$ `=` $E$ `%>` directive defines variable $x$ in the UAS, and assigns to $x$ the result of evaluating the expression $E$. The expression $E$ can be a JavaScript expression, a source-specific language (such as a SQL query in the case of relational database sources) or a JSON++ path. For example, the template of our running example (lines 6-8) employs a `let` directive to create a variable `readings` containing all temperature readings (retrieved from a relational DBMS through a SQL query).

**Reporting syntax and semantics.** Computation in a template is specified using the `print` and `for` directives.

| | | | |
|---|---|---|---|
| 1 | *template* | → | `<% template` *template_name* (*param_list*) `%>`<br>*let**<br>*unit*<br>`<% end template %>` |
| 2 | *param_list* | → | ( *var_name* (, *var_name*)* )? |
| 3 | *unit* | → | `<% unit` *unit_class* `%>`<br>*value*<br>`<% end unit %>` |
| 4 | *value* | → | *jsonpp_value* |
| 5 | | \| | *unit* |
| 6 | | \| | *print* |
| 7 | | \| | `[` *for* `]` |
| 8 | | \| | `<` *for* `>` |
| 9 | | \| | *if* |
| 10 | | \| | *bind* |
| 11 | | \| | `{` *event**<br>(`"`*string*`" :` *value*<br>(`, "`*string*`" :` *value*)* )? `}` |
| 12 | *let* | → | `<% let` *var_name* `=` *expr* `%>` |
| 13 | *print* | → | `<% print` *expr* `%>` |
| 14 | *for* | → | `<% for` *var_name* `in` *expr* `%>`<br>*let**<br>*value*<br>`<% end for %>` |
| 14 | *if* | → | `<% if` *expr* `%>`<br>*value*<br>(`<% elif` *expr* `%>`<br>*value*)*<br>(`<% else %>`<br>*value*)?<br>`<% end if %>` |
| 15 | *bind* | → | `<% bind` *var_name* = *expr* `%>` |
| 16 | *event* | → | `<% event` *event_name action_name* `%>` |
| 17 | *expr* | → | *js_expression* |
| 18 | | \| | *source_expression* |
| 19 | | \| | *json_path* |

Figure 3: BNF Grammar for Templates

The `<% for` $x$ `in` $E$ `%>` $B$ `<% end for %>` directive specifies that variable $x$ iterates over the result of the expression $E$. In each iteration, the body $B$ of the `for` loop is instantiated. For example, the template of our running example (lines 14-19) uses a `for` directive to iterate over the sensor readings (stored in the `readings` variable). For each reading, it generates a new JSON tuple of the form `{y:..., color:...}`, which is the data format expected by the HighCharts unit for each data point.

The `<% print` $E$ `%>` directive instantiates the result of expression $E$. For example, the template of the running example uses two `print` directives to generate the values of the `y` and `color` attributes of each JSON tuple produced by the `for` directive. *YannisP: The part between the "{}" could be removed* {The value of `y` is created by printing the value of the `reading` variable (line 16), while the value of `color` is generated by calling the `toCSS` function, which takes as input the current reading and the normal temperature (set through the slider) and produces a CSS color code according to the coloring schema explained above (line 17).}

**Collecting data.** In addition to specifying how to compute a template instance, the template's `bind` directive allows the developer to specify how data are collected from user input on the page.

The `<% bind` $x$ = $E$ `%>` directive specifies that the template instance attribute value in whose position the directive appears will be assigned to variable $x$. This allows UAS variables to become bound to input received by visual units. *YannisP: If UAS has not been defined we need to say what is the JavaScript target.* *Costas: We should define a Model/VDB variable, that encompasses every client-side variable*

```
<% let age_groups =
   SELECT agegroup, count(*) AS total
   FROM (SELECT CASE
    WHEN age BETWEEN 0 AND 9 THEN '0 to 9'
    WHEN age BETWEEN 10 and 19 THEN '10 to 19'
     ...
    FROM (SELECT * FROM page_views pv join visitors v
          on pv.v_id = v.vid where time BETWEEN
          <%=min_time and <%=max_time) joined_data) jd
   GROUP BY agegroup
   ORDER BY agegroup ASC %>
```

(a) Data retrieval

```
age_groups = [
   {age_group: '0 to 9', total: 12},
   {age_group: '10 to 19', total: 67},
   {age_group: '20 to 29', total: 84},  ...]
```

(b) Query Result

```
<% unit highcharts %>
{
  title: 'Visitor information',
  xAxis : {
    labels : [
      <% for v in age_groups %>
        <% print v.age_group %>
      <% end for %>]
  }
  series: [{
    data: [ <% for v in age_groups %>
        {
          y  : <% print v.total %>
        }
      <% end for %> ]
  }]
}
<% end unit %>
```

(c) Template `temp_view`

Figure 4: Template, template instance, and UAS configuration file for the running example

*used in an application. The target in this case would be the Model.x variable* For instance, the template of our running example (line 30) uses a **bind** directive to assign to variable `threshold` the current value of the slider (which is returned by the slider unit as the value of the `value` variable). The **bind** directive also allows the developer to specify an expression $E$, whose value will be assigned to the variable when the template is first instantiated. For example, the slider of our running example is initialized with the value 65.

## 5. DISCUSSION

DISCUSSION

## 6. CONCLUSION

## 7. REFERENCES


[1] Angular leaflet directive. http://tombatossals.github.io/angular-leaflet-directive/.

[2] AngularJS. http://angularjs.org/.

[3] Apache mahout. https://mahout.apache.org/.

[4] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with datahub. *Proceedings of the VLDB Endowment*, 8(12):1916–1919, 2015.

[5] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[6] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD*, pages 61–71, 1986.

[7] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1121–1128, 2009.

[8] Catel. http://www.catelproject.com/.

[9] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[10] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[11] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[12] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment*, 8(12):2024–2027, 2015.

[13] D3: Data-driven documents. https://d3js.org/.

[14] A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. In *Workshop on Algorithms and Data Structures*, pages 265–276. Springer, 1993.

[15] M. Derthick, J. Kolojejchick, and S. F. Roth. An interactive visual query environment for exploring data. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 189–198. ACM, 1997.

[16] Ember. http://emberjs.com/.

[17] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with Strudel. *VLDB J.*, 9(1):38–55, 2000.

[18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[19] Y. Fu, K. Kowalczykowski, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Ajax-based report pages as incrementally rendered views. In *ACM SIGMOD*, 2010.

[20] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The sql-based all-declarative forward web application development framework. In *CIDR*, pages 69–78, 2011.

[21] ggvis: Interactive grammar of graphics for r. http://ggvis.rstudio.com/.

[22] Google maps api, 2009. http://code.google.com/apis/maps/.

[23] A. Gupta and I. S. Mumick. Maintenance of

materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[24] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD*, pages 157–166, 1993.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[26] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library: Or mad skills, the sql. *Proc. VLDB Endow.*, 5(12):1700–1711, Aug. 2012.

[27] highcharts. http://www.highcharts.com/.

[28] highcharts-ng. https://github.com/pablojim/highcharts-ng/.

[29] N. Kamat, E. Wu, and A. Nandi. Trendquery: A system for interactive exploration of trends. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 12:1–12:4, New York, NY, USA, 2016. ACM.

[30] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing ids to accelerate incremental view maintenance. In *ACM SIGMOD Conference*, pages 1985–2000, 2015.

[31] Knockout. http://knockoutjs.com/.

[32] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[33] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, Aug. 2016.

[34] Leafletjs. http://leafletjs.com/.

[35] E. Liarou and S. Idreos. dbtouch in action database kernels for touch-based data exploration. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1262–1265, 2014.

[36] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: integrated querying and visual exploration of large datasets. In *ACM SIGMOD Record*, volume 26, pages 301–312. ACM, 1997.

[37] Mllib. http://spark.apache.org/mllib/.

[38] Mvvm light. http://www.mvvmlight.net/.

[39] Mvvmcross. https://mvvmcross.com/.

[40] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[41] React. https://facebook.github.io/react/.

[42] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2017, 2015.

[43] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016.

[44] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. G. Parameswaran. zenvisage: Effortless visual data exploration. *CoRR*, abs/1604.03583, 2016.

[45] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[46] Vega, a visualization grammar. http://trifacta.github.io/.

[47] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.

[48] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[49] Y. Wu, J. M. Hellerstein, and E. Wu. A devil-ish approach to inconsistency in interactive visualizations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 15:1–15:6, New York, NY, USA, 2016. ACM.

[50] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *ICDE*, page 32, 2006.

[51] K. Zoumpatianos, S. Idreos, and T. Palpanas. Rinse: interactive data series exploration with ads+. *Proceedings of the VLDB Endowment*, 8(12):1912–1915, 2015.