Jupyter on Steroids

ABSTRACT

This abstract is way too long. Most of it should be in the introduction. We should try to keep it under half column with a flow like: issue -> "This paper proposes..." -> Results if any.

A plethora of tools and frameworks has been proposed, by the database and systems community, for streamlining the admittedly arduous tasks of data retrieval, data curation and data visualization. Since these tasks, typically, have to be carried out in most data science projects, a significant portion of the afore-mentioned tools, focuses on providing a simple-to-use way for performing them. Despite the very promising results, however, these tools often seem to limit flexibility as they only focus either on a predetermined set of use-cases or on only a small fraction of tasks of a typically much bigger pipeline. This lack of flexibility often pushes code-literate data scientists to the tested and proven path 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. To address these issues, we propose a set of extensions on interactive notebooks...

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

DOI: 10.1145/1235

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.

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 6 provides further discussion regarding our proposed extension and presents other useful aspects of it not used in our walkthrough. Finally, Section 7 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. It enables the analyst to write inlined, 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 needs to read lengthy documentation pages and stackoverflow posts 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.

Costas: Some of these steps, mentioned up to here are common for the data retrieval and visualization stages (e.g installing database drivers and visualization libraries, reading documentation pages to figure out how to use the and so on. Can we group them into a single section for ipython and then explain how things differ in ViDeTTe?

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 that 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, this allows the notebook user to get a glimpse of that information when generating the queries.

Once this step is completed, the analyst is free to issue queries in order to access database systems. Figure **??** 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 1c 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

```
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',
  type: 'line'
  xAxis : {
    labels : [
      <% for reading in readings %>
        <% print reading.time %>
      <% end for %>],
    min : <% bind min_time %>,
    max : <% bind 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

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 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 that describes the visualization, namely, unit instance. The visual unit internally uses the unit instance to invoke the appropriate renderer calls that will generate the expected visualization. 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 the unit instance.

4.2 Template

Costas: We should make it clear in the intro that ViDeTTe operates on a JSON datamodel. This means that source wrappers, visual units and templates all operate on the same

```
<% 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
          <% print min_time %> and
          <% print 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 2: Template, template instance, and UAS configuration file for the running example

model, no conversions are required from one to another. Additionally. since our datamodel is plain-old javascript objects, the analyst does not need to learn a new API in order to interact with our values.

Templates are declarative specifications that can be used to generate ViDeTTe variables or unit instances. The template language supports a set of *template directives*, all of which operate on the JSON datamodel. These directives are used to describe computation, define variables and set up data collection. Due to lack of space we do not include a figure containing a formal BNF grammar for the template language, instead we simply describe each of them in detail and provide a concrete example that illustrates their use.

Defining variables. A template may define variables that are added to the notebook's environment so that they can be used in subsequent computation. The **`<% let`** x **`=`** E **`%>`** directive defines variable x and assigns to it the result of the expression E. E can denote three types of expressions: path navigation on JSON data, invocation of a python function that performs computation by using as input and output JSON values or a source-specific language (such as a SQL query). For instance, the template shown in Figure 1b employs a **`let`** directive to create a variable `readings` containing the visitor information that will be displayed in the chart (retrieved from a relational DBMS through an SQL query).

Reporting syntax. Value assignments and iterations over collections are specified by using the **`print`** and **`for`** directives. The **`<% print`** E **`%>`** directive evaluates the expression E and returns the result, while the **`<% for`** x **`in`** E **`%>`** B **`<% end for %>`** directive specifies that variable x iterates over the result of E and, in each iteration, it instanciates the body B of the **`for`** loop. In Figure 1e, in lines 7-9 and 14-18 a **`for`** directive is used to iterate over the readings retrieved from the database and for each reading, it generates a new JSON value, with the use of the **`print`** directive. Particularly, in line 8 the template generates a string (the time label), which is added to the labels array (which contains the labels that will appear on the x axis of the chart). In lines 15-17 it generates a JSON object of the form {y: ...} and adds it to the data array. The data array in highcharts contains the points that will appear in the chart.

Collecting data. In addition to specifying how to compute a template instance, the template's **`bind`** directive allows the analyst to specify user input collection. Specifically, the **`<% bind x %>`** directive describes a two-way bind that will be created between the part of the unit instance appearing on its left side and to variable x. For instance, in Figure 1e in lines 10-11 we create a two-way binding between the min and max boundaries of the chart and the min and max values of the time labels that have been retrieved (namely min_time and max_time). While the user interacts with chart he can select a particular region in the chart which in turn updates the values *min_time* and *max_time*. This event invokes an internal propagation algorithm (which will be described in the next section) which triggers the revaluation of the ViDeTTe statements that use the values *min_time* and *max_time* in other parts of the notebook.

5. INTERACTIVE INTERFACE

As the notebook reader interacts with a visualization the underlying visual unit triggers actions, which make the visualization react to the reader's input. For instance, as the user selects a particular region of the chart (shown in figure *Costas: ADD FIGURE*) by dragging and dropping the mouse over that particular area, she causes the visualization to zoom into the selected area. This behavior is dictated by the underlying visual unit which listens for events (or rather series of events) and actively causes mutations to the visualization. Other than mutating the visual layer, these units also mutate the unit state. In this particular case, zooming into a region, causes the min and max attributes of the chart's visual state (shown in Figure 1d, in lines 5-6) to mutate accordingly. If the template language contains variables bound to these parts of the unit instance, ViDeTTe propagates mutations to the respective variables. Furthermore, if these variables are used in other parts of the notebook, a change propagation algorithm identifies all the statements that depend on them and triggers their re-evaluation.

For instance, the template shown in Figure 1e contains two variables, namely *min_time* and *max_time*, which are bound to the *min* and *max* unit instance attributes respectively. These variables are later used in the parameterized query shown in Figure 2a which retrieves information about the users that visited the website in the time-frame specified by *min_time* and *max_time*, groups the result in age groups and sums up the visits made by each age group. Figure 2b, contains the result of that query. This result is assigned to variable *age_groups* which is then used to produce the bar

ALGORITHM 1: Change Propagation Algorithm

```
1  function change-propagation(Notebook $N$, Environment env, Set of Input Diffs $D$)
2      for each statement $s$ in each block of $N$ do
3          if $s$ is <% let $x$ = $E$ %> and $E$ contains
4          directive: <%for $v$ in $E''$%> $B$ <%end for%>
5          or directive: <% print $E'$ %> then
6              if There is $\Delta(t;p) \in D$ with $t$ that matches $E'$ or $E''$ then
7                  Reevaluate $E$ and assign result $r$ to $x$;
8                  Update env with new value of $x$;
9                  Construct $\Delta(x;r)$ and add it to D;
10         else if $s$ is <% let $x$ = $E$ %> and $E$ contains
11         directive: <%for $v$ in $E'$%> $B$ <%end for%> with $B$ containing nested directives <%for $v$ in $E''$%> or <%print $E'''$%> then
12             if There is $\Delta(t;p) \in D$ with $t$ that matches $E''$ or $E'''$ then
13                 Reevaluate $E$ and assign result $r$ to $x$;
14                 Update env with new value of $x$;
15                 Construct $\Delta(x;r)$ and add it to D;
16         else if $s$ is <%unit $U$%> $B$ <%end unit%> and $B$ contains directives <%for $v$ in $E''$%> or <%print $E'$%> then
17             if There is $\Delta(t;p) \in D$ with $t$ that matches $E'$ or $E''$ then
18                 Reevaluate $E$ and invoke unit U with result $r$ ;
19     end
20     return $D_{out}$;
```

chart appearing in Figure Costas: add Figure (the template shown in Figure 2c shows the template that created that chart)

5.1 Data Model of Change Propagation

In this subsection, we will define the propagation algorithm that triggers this interactive behavior. We will also define the internal data model used by both visual units and the propagation algorithm to describe mutations on ViDeTTe variables.

Diffs. Since, as explained above, both ViDeTTe variables and template instance are represented using JSON, changes to either of them are represented as diffs to JSON values. A *diff* to a JSON value is of the form $\Delta(\hat{p})$, where $\hat{p}$ is the path to the element that is being modified. The first token of that path p denotes the variable that is being mutated, while the remaining set denotes the individual nested attribute or array element. Specifically, in order to describe an individual attribute k of a JSON value identified by path p we use the notation $p.k$ and in order to describe the i-th element of the array targeted by a path p we use the notation $p[k]$.

5.2 Change Propagation Algorithm

Algorithm 1 summarizes the change propagation module, that takes as input a set of diffs generated by visual units, as a result of the user interaction and triggers the reevaluation of all statements that depend on the targeted variables. Specifically, the algorithm iterates over each statement of each block that appears in the notebook (line 2)

6. DISCUSSION

DISCUSSION

7. CONCLUSION

8. 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.