ViDeTTe Interactive Notebooks

Konstantinos Zarifis
University of California, San Diego
zarifis@cs.ucsd.edu

ABSTRACT

Interactive notebooks allow the use of popular languages, such as python, for composing data analytics projects. The interface they provide, enables data scientists to import data, analyze them and compose the results into easily readable report-like web pages, that can contain re-runnable code, visualizations and textual description of the entire process, all in one place. Scientists can then share such pages with other users in order to present their findings, collaborate and further explore the underlying data.

However, as we show in this work, interactive notebooks lack in interactivity and ease of use, both for the data scientist that composes the notebook and, even more so, for the reader of the resulting notebook. Specifically, while the user interface allows readers to rerun or extend the code included in a notebook, it does not allow them to directly interact with the generated visualizations in order to trigger additional computation and further explore the underlying data. This means that only code-literate readers can further interact with and extend such notebooks, while the rest can only passively read the provided report. This comes in stark contrast to OLAP data cube interfaces, which utilize the user interaction to trigger additional data exploratory capabilities. Adding OLAP-like reactive functionality in notebooks further increases the required technical expertise as event-driven logic (which is typically required in fully-fledged web applications) has to be explicitly added by the data analyst.

To address these issues, we propose ViDeTTe, an engine that enhances notebooks with capabilities that benefit both data scientists and non-technical notebook readers. ViDeTTe introduces a declarative language that simplifies data retrieval and data visualization for analysts. The generated visualizations are capable of collecting the reader's input and reacting to it. As the user interacts with the visualizations, ViDeTTe identifies subsequent parts of the notebook that (can describe additional computations and) depend on the user's input and causes their reevaluation. By doing this, ViDeTTe offers enhanced data exploratory capabilities to readers, without requiring any coding skills, while at the same time lowering the technical expertise needed for the development of reactive notebooks.

KEYWORDS

ACM proceedings, LaTeX, text tagging

WOODSTOCK'97, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

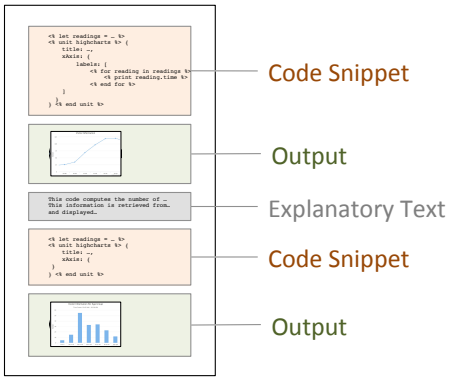

Abbildung 1: Notebook interface

ACM Reference Format:
Konstantinos Zarifis. 1997. ViDeTTe Interactive Notebooks. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

The database, HCI and systems communities have proposed tools that simplify common tasks of analytical processes. Such tools, however, either focus on only individual stages (such as data retrieval or visualization) of a much bigger analytical pipeline or they focus on particular use cases (for instance pattern matching, finding correlations and so on). Despite their remarkable success, data analysts often want the flexibility that comes from combining diverse tools and computation libraries in order to perform an analysis. This need often pushes code-literate data analysts to the tried-and-true interactive notebooks, such as Jupyter [**Jupyter**].

Interactive notebooks allow the use of popular, highly expressive, imperative languages, such as Python, for describing tasks such as data retrieval, processing and visualization, all in one platform. Due to the popularity of the languages they support, there is a massive collection of third-party utility libraries that can be used to facilitate such tasks. Furthermore, the web environment of interactive notebooks enables collaboration between data analysts, since it allows them to develop and run code that processes data and generates visualizations directly on the browser. Lastly, after completing an analysis, data analysts can compose their findings into an interactive report-like page, that contains re-runnable code, visualizations and textual description of the analysis, which can also be shared with non-technical users. Figure 1, shows the interface of Jupyter notebooks. It comprises a sequence of blocks that are created "on-demand"by the data analyst, each block can contain explanatory text or re-runnable coding snippets that once evaluated output the result (which could be a visualization or plain text) into the subsequent block.

Page Views					
id	vid	url	time	date	revenue

Visitors					
vid	name	lastname	username	age	gender

Tabelle 1: Schema description of the two tables in our database.

Jupyter notebooks, however, have limitations that impact the exploratory capabilities that can be performed by the potentially non-technical, readers of the published notebooks. Particularly, while notebooks can contain visualizations that showcase important aspects of a data analysis, the readers cannot interact with them in the same way they could if these visualizations were part of a typical OLAP dashboard application. Specifically, even if the analyst that composes the notebook uses third-party web-based visualization libraries that support user interaction, the default side-effect of this interaction is to only cause local changes to the visualization [**Bokeh**, **plotly**, **ipyvega**, **Altair**] the reader interacts with and, therefore, do not trigger additional data retrieval, computation or mutations to other visualizations that are included in the notebook (which are typically generated by subsequent coding blocks). For this reason, only code-literate readers, capable of extending the notebook with more coding snippets, are able to further explore the underlying data (or examine additional hypotheses about that data), while the rest are limited to passively reading the generated notebook.

In order to add such OLAP-like functionality to notebooks, a data analyst has to explicitly implement event-driven logic for every UI event that might occur. This process however is not trivial, manually specifying elaborate side-effects for each event that might occur is an extremely arduous and error-prone task, especially, when multiple events cause mutations to the same visualizations of the notebook. This coding style is used by web developers (in MVC frameworks) when building fully-fledged applications, and it typically requires advanced skill-set that data analysts might lack. At the same time, such event-driven programming style is not typically used in Jupyter notebooks. Lastly, note that the provided logic executed after each event may need to mutate visualizations generated by subsequent coding blocks, this functionality however, is not supported by most third-party visualization libraries.

Contributions. In order to resolve these issues, we extend Jupyter interactive notebooks with the ViDeTTe notebook engine. The main contributions of this extension are: (a) visual constructs, capable of collecting the reader's input as they interact with the produced visualizations, (b) A declarative template language that can (optionally) be used to declaratively describe computation, data transformations and user input collection, (c) A propagation algorithm, capable of inferring dependencies and propagating changes between visual constructs and described computations, thus introducing a truly reactive behavior to notebooks. These reactive capabilities, enable non-technical, code illiterate readers to further explore the underlying data without requiring complex event-driven logic from the data analysts that composed the notebooks.

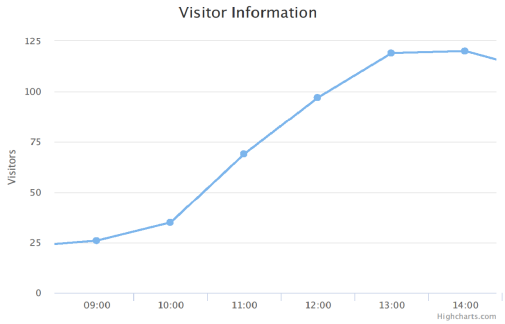

Abbildung 2: Line chart showing visitors per hour

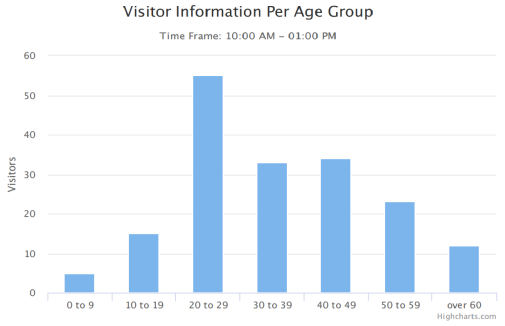

Abbildung 3: Bar chart showing age groups of visitors

2 RUNNING EXAMPLE

In order to illustrate the issues with existing notebooks and describe the extensions in ViDeTTe we will use the following example of a potential analysis:

Example 2.1. *Consider a data analyst, working for a news portal. The analyst wishes to convince the lead editor of the portal that by publishing more articles and advertisements that pertain to particular audiences during specific time-windows of the day, they can maximize the portal's revenue. In order to achieve this, the analyst intends to obtain and plot the number of readers that visit the website during the day; then for various time-windows she would like to obtain information about the readers' demographics (for instance, the visitors' age groups) and plot the result. Lastly, she would like to compute the portal's actual and predicted revenue (using a linear regression model) and plot the the results, side-by-side, thus illustrating whether there is room for improving revenue.*

For this analysis, we assume that the news portal maintains data about its reader-base in a Postgres database. Table 1 shows a potential schema, that could be used for storing visitor information. The database contains two tables, namely "Page Viewsänd "Visitors"; table "Visitors"contains information about each reader, such as the reader id, name, lastname, username, age and gender. The table "Page Views"maintains a tuple for each visit, and it consists of a visit id, the visitor id (foreign key referencing the visitor), the url of the visited page, the hour and date in which the user visited the website and the revenue that was collected during this visit. The reader information could have been retrieved, with their permission, from various social media services (such as Facebook, Google etc). In order to construct this notebook, the data analyst must (a)

retrieve website access information from the database, by joining the two tables on the visitor id (b) generate a plot that shows the number of users that visit the website during the day, (c) issue a query that counts the number of visitors per age group, for each potential set of selected hours, (d) create a bar chart that shows the number of visitors per age group, (e) use an already trained predictive model in order to predict the expected revenue for the selected time window and (f) plot a bar chart, showing the actual and the predicted revenue generated in selected time window.

3 DATA ANALYSIS STEPS

We will now describe how a data scientist would perform this analysis with a traditional interactive notebook that uses Python. We will also illustrate how the limited exploratory capabilities of the produced analysis impacts the extraction of useful information.
Performing Data access, data conversions and visualizations. In order to retrieve website access information from the database, the data analyst, needs to employ third-party Python drivers. After establishing the connection with the database system, and issuing the queries, the analyst, is able to consume the results using the internal, to the library, data types. Afterwards, the analyst needs to perform (schema) conversions in order to invoke the respective visualization library, with the appropriate arguments, thus constructing the first visualization.

The next step is for the analyst to issue a query that counts the visitors per age group for a set of selected hours and construct a bar chart showing the result. The obtained dataset will also go through a machine learning package (for instance, the scikit-learn package [**scikit-learn**]), that employs a linear regression model to predict the expected revenue during the selected hours. The predicted revenue will be plotted in a new graph next to the actual revenue produced during these hours. Note that selecting a meaningful time frame, is of utmost importance, because it will affect all the remaining steps of this analysis, and ultimately the insight, the portal editor will gain, about potential ways to increase the revenue.
Limited interactive exploratory capabilities. Unfortunately, there is no straightforward way for selecting a meaningful time frame. The analyst will have to go through a process of trial and error, by issuing an arbitrary number of such queries and plotting the results, until she finds a set that produces valuable insight. Furthermore, even if the analyst goes through this process for the data corresponding to a particular day, the time frame selection they made, might not produce any valuable insight for the dataset corresponding to another day. Most importantly the non-technical readers cannot use the notebook to test similar hypotheses, they are limited to passively reading the ones that were hardcoded by the analyst. This aspect of introducing the human in the loop is improved dramatically with ViDeTTe notebooks.

4 REACTIVE NOTEBOOKS

Ideal reactive behavior and functionality. Co-dependent reactive charts are tremendously useful in this scenario. If the analyst was able to introduce a chart that allows the reader to select a particular range of hours, by directly interacting with it, and use that input to retrieve and plot only the user demographics for this particular range, she could enable the readers to further explore the underlying data without requiring any code literacy from them. Figure 4 graphically depicts this process (note that the charts that appear side-by-side in Figure 4 actually appear after the respective coding blocks that generated them in the notebook). The first figure shows the reader of the notebook, selecting a particular time frame; this causes the line chart to zoom in, thus showing only the selected time frame. After the reader performs the selection on the first chart, the subsequent bar chart is also updated thus only showing the age groups of the users that visited the portal during the selected hours. This feature adds useful exploratory capabilities to the notebook, which is of great value to readers. It enables them to further analyze the underlying dataset used by the notebook by simply interacting with the provided visualizations. (For a live demonstration that illustrates how this analysis can be performed using ViDeTTe, visit the page: https://czarifis.github.io/vidette-prototype/)

This kind of reactive behavior cannot be easily expressed in traditional interactive notebooks. It is important to note that implementing a reactive behavior that can collect user input and use it to access additional data or mutate existing visualizations is extremely arduous currently in interactive notebooks. It is identical to the process developers go through when implementing full blown web applications. It requires more time and effort as well as technical expertise (with third-party web components or input widgets) that data analysts, might lack. More specifically, describing reactive behavior entails the implementation of actions (functions) that have to be invoked when particular mouse events take place on the pixels of the browser that correspond to the visualization. These actions describe the side-effects that will by applied when such events occur. The developer must install observers that listen for such events and then provide the imperative application logic that asynchronously accesses back-end databases to retrieve new data based on the user's selection, process them appropriately and cause mutations to the respective visualizations that depend on them. Most importantly, in some cases the events have to take place in a particular order for the side-effects to make sense (for instance, range selection requires mousedown-mousemove-mouseup). In such scenarios, the developer's logic has to share state between these actions and accordingly modify it when certain events occur. Implementing the code that deals with such cases is extremely error-prone (it is known as callback hell)

5 VIDETTE NOTEBOOKS

ViDeTTe notebooks support reactive behavior using a different approach. Instead of forcing analysts to provide logic that explicitly describes the side-effects (which typically include data access, processing and visualization steps) for each event that can take place, it allows them to implicitly describe them as a view-over-data. The advantage of this approach is that, when mutations occur at the underlying data (for instance, as a result of user interaction), the ViDeTTe engine can automatically infer which parts of the notebook code are affected and automatically cause their reevaluation. In order to achieve this, ViDeTTe provides constructs, called **visual units**, that not only streamline the creation of visualizations given data, but most importantly they streamline the collection of user input from said visualizations. As readers interact with visualizations, the respective visual units cause mutations to the underlying data. Given such mutations ViDeTTe identifies the coding statements

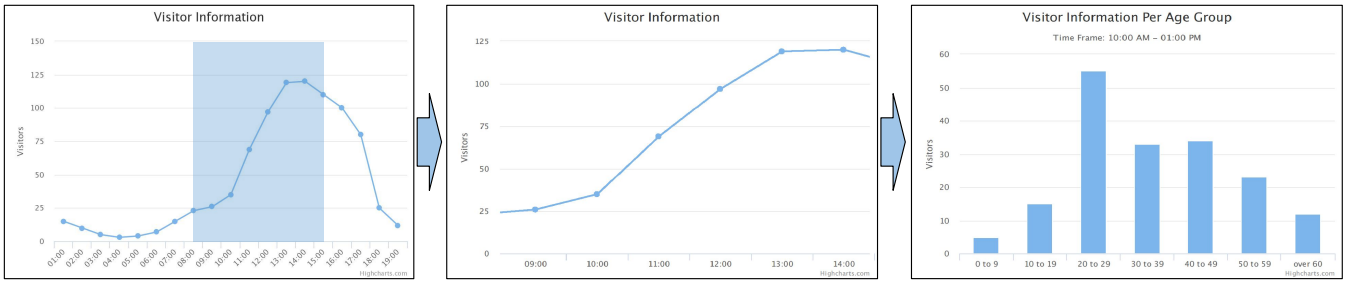

Abbildung 4: Demonstration of reactive charts. The reader's selection automatically updates both charts.

```
<% 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 %>;
```

(a) Data retrieval

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

(b) Unit with evaluated unit state

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

(c) Query Result

```
<% 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 %>
```

(d) Template generating unit state of first chart

Abbildung 5: Declarative data retrieval, evaluated unit state and Template describing unit state for running example

that depend on the mutated data and causes their reevaluation, thus automatically updating the state of (including the dependent computations and visualizations that appear on) the notebook.

In order to identify coding statements that depend on mutated variables, ViDeTTe uses a control flow analysis. However, since python is a dynamically-typed language, the employed analysis can often result in false positives, thus causing the reevaluation of statements that are not actually affected by mutations. In order to avoid unnecessary reevaluations, the data analyst can choose to explicitly define the dependent Python statements using annotations. Alternatively they can use a domain specific **declarative template language**, provided by ViDeTTe. When this template language is used, ViDeTTe can automatically identify the dependent expressions without false-positives. This template language can be used for the invocation of system-provided functions that describe data access (namely, **source wrappers**) and invocation of reusable Python functions (that can describe an unbounded number of computations). Also, the template language can describe schema conversions between visual units and source wrappers.

Source wrappers. Source wrappers allow the use of source specific languages for querying database systems (for instance SQL can be used to query MySQL and PostgreSQL databases, N1QL for Couchbase etc.,). When using a ViDeTTe notebook, data analysts can directly issue queries from the notebook interface. This query is directed to the appropriate source wrapper, that evaluates it, propagates it to the respective data base system and assigns the result to a Python variable. This variable can later be used in subsequent parts of the notebook (as will be described later in this section). Figure 5a contains the first query of the analysis that retrieves the dataset required to plot the chart shown in Figure 2. Specifically, 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 and assigns the result the *readings* variable. Once the respective active function evaluates the query, it obtains and automatically parses the result into a Python array of dictionaries (used to maintain the tuples), as shown in Figure 5c. Note, that the analyst no longer needs to waste any time reading documentation pages that describe how to issue queries and parse the results, all these steps are handled automatically by the respective source wrapper.

Visual Units. Visual units are constructs capable of generating fully reactive visualizations, that take advantage of the advanced interactive capabilities of modern browsers. In the eyes of an analyst a visual unit is simply a black box that takes as input a value that describes the state of the visualization. This value is called *unit instance*. The visual unit internally uses the unit instance to invoke the appropriate renderer calls that will generate the expected visualization, thus absolving the data analyst from having to

perform these tasks manually. 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 Python value corresponding to the unit instance. Figure 5b shows the unit instance of type `highcharts`, that once added to a coding block generates the visualization shown in Figure 2. The unit instance describes all the information that will be displayed in the visualization (such as the type and 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. As the reader interacts with visualizations, they can trigger mutations in the unit instance. For example, if the reader selects a particular area of the chart generated by the unit instance shown in Figure 5b, the min and max attributes (in lines 6-7) will be updated accordingly.

Templates. Despite the fact that both source wrappers and visual units use Python variables, data scientists often need to perform additional computation or data transformations. In order to facilitate this process, without requiring the use of imperative logic, ViDeTTe provides a declarative template language that operates on Python variables. Note, that analysts are not forced to use this language, if they don't wish to, they can simply manually construct the same Python variables (using Python code) and invoke the appropriate visual units or source wrappers by providing the variables as arguments. The template language supports a set of *template directives*, all of which operate on Python variables and simplify the task of creating nested data. Due to lack of space we do not include a formal definition of the grammar, 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 statements. The **<% let** *x* **=** *E* **%>** directive defines variable *x* and assigns to it the result of the expression *E*. *E* can denote four types of expressions: (a) path navigation on nested Python variables, (b) invocation of a Python functions, (c) another subexpression containing directives that report data (defined in the next subsection) which generate nested values or (d) a source-specific language (such as an SQL query). For instance, the template shown in Figure 5a 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 data. 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. Specifically, in Figure 5d, in lines 6-8 and 13-17 a **for** directive is used to iterate over the readings retrieved from the database and for each reading, it generates a new value, with the use of the **print** directive. Particularly, in line 7 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 14-16 it generates a Python dictionary 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.

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 coding snippet of N do
3          if s is <% let x = E %> then
4              if There is Δ(t) ∈ D with t that targets any
               subexpression of E then
5                  Reevaluate E and assign result r to x;
6                  Update env with new value of x;
7                  Construct Δ(x) and add it to D;
8          else if s is <%unit U%> B <%end unit%> then
9              if There is Δ(t) ∈ D with t that that targets any
               subexpression of B then
10                 Reevaluate B and invoke unit U with result r ;
11         else if s is Python statement then
12             if There is Δ(t) ∈ D with t that that targets any
               subexpression of s then
13                 if s is an assignment x = E then
14                     Reevaluate E and assign result to x;
15                     Update env with new value of x;
16                     Construct Δ(x) and add it to D;
17                 else
18                     Reevaluate s ;
19     end
```

Invoking Python Functions. A template can also describe the invocation of Python functions, thus allowing any arbitrary computation. For instance, the snippet shown in Figure 6f (which generates the bar-chart illustrating the predicted and actual revenue) in lines 5 and 6 invokes two functions: the *predict_revenue*() and *sum_revenue*() by using the **print** directive while providing as input an array of dictionaries that is initialized in Figure 6a; function *predict_revenue*() (shown in 6c), inputs the array into a trained linear regression model and returns the revenue prediction, while function *sum_revenue*() (shown in 6d), simply computes the actual revenue.

Collecting data. The template's **bind** directive allows the analyst to specify user input collection. Specifically, the **<% bind x %>** directive describes a two-way binding between the part of the unit instance appearing on the left side of the directive and the variable *x*. Once the reader interacts with the generated visualization and causes mutations to the underlying unit instance, these mutations will be propagated to the respective bound variables. For instance, in Figure 5d in lines 9-10 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 received from the database (namely min_time and max_time). As the reader interacts with the generated chart she can select a particular time frame by dragging and dropping the mouse over a region. This action updates the min and max attributes of the unit instance which in turn updates the values *min_time* and *max_time*.

Internal Data Model & Propagation Algorithm. Once a set of variables is mutated by a visual unit as a result of reader interaction, or by a source wrapper as a result of a push-event from an underlying

```
<% let visits =
   SELECT * FROM page_views pv join visitors v
   on pv.v_id = v.vid where time BETWEEN
   <% print min_time %> and <% print max_time %>;
<% 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  visits
   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

```
def predict_revenue(input):
  y_pred = LinearRegression.predict(input)
  return sum(y_pred)
```

(c) Python function predicting revenue

```
def sum_revenue(input_list):
  return sum(item['revenue'] for item in input_list)
```

(d) Python function summing actual revenue

```
  <% unit highcharts %>
  {
    title: 'Visitor information',
    type: 'bar',
    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 %>
```

(e) Template generating unit state of second chart

```
<% unit highcharts %>
{title: 'Predicted and Actual revenue',
  ...
  data: [
    {y : <% print predict_revenue(visits) %> }
    {y : <% print sum_revenue(visits) %> }
  ]...}
```

(f) Actual and Predictive revenue visualization

Abbildung 6: Declaration of "reactive"parameterized view, data visualization specifications and Python function declaration and invocation

database (in cases when the analyst visualizes live data), ViDeTTe is responsible for reflecting these changes to all other parts of the notebook that depend on those variables. For instance, in Figure 6a we show a parameterized query that collects all users that visited the website in the time-frame specified by *min_time* and *max_time*, categorizes them into age groups and lastly counts the users that correspond to each age group. The result of this query is assigned to the variable *age_groups* (Figure 6b shows the contents of this variable). The variable *age_groups* is then used in the template shown in Figure 6e (lines 7-9 and 12-16) to produce the bar chart appearing in Figure 3. In this scenario, if the reader's interaction with the chart shown in Figure 2, caused the mutation of variables *min_time* and *max_time* (note that the values *min_time* and *max_time* where collected by the visual unit shown in Figure 5d, lines 9 and 10), then ViDeTTe must trigger a chain reaction that causes the reevaluation of both the query and the following visualization in order to reflect the changes. We will now describe this process.

Representing mutations through Diffs. Once a variable is mutated, the respective visual unit or source wrapper produces a diff that describes this mutation. A *diff* is the internal data model that is shared between source wrappers, visual units and the change propagation algorithm, this data model is hidden from data analysts and notebook readers. A diff that targets a nested value is of the form $\Delta(\hat{p})$, where $\hat{p}$ is the path to the value that is being modified.

Change Propagation Algorithm.

Once a diff is generated for each mutated variable ViDeTTe invokes the change propagation algorithm, shown in 1. The algorithm takes as input a set of diffs, and only invokes the reevaluation of statements that depend on the targeted variables. Specifically, it iterates over each statement of each coding block that appears in the notebook (as shown in line 2) and performs the following checks:

- If the statement defines a new variable x (line 3), then the algorithm visits every sub-expression of the **`let`** directive and checks if it is targeted by any diff in D (line 4). If there is, the change propagation algorithm causes the reevaluation of the **`let`** statement (line 5), updates the notebook environment with the new value of x (line 6) and lastly constructs a new diff that describes the mutation to the variable x (line 7). Note that the Expression shown in Figure 6b is an SQL statement that contains subexpressions (parameters), so given a diff of type: $\Delta(\hat{min\_time})$ (which could be created as the reader interacts with the chart), the propagation engine identifies that there is a subexpression targetted by that diff thus causing the reevaluation of the source wrapper, resulting in additional data.
- If the statement constructs a visualization using a visual unit of type U with body B (line 8), then the algorithm visits every sub-expression in body B and checks if it is targeted by any diff in D (line 9). If there is, the change propagation algorithm causes the reevaluation of the entire body B, thus generating the unit instance r and then reinvokes the unit U with new unit instance (line 10).
- If the statement is a typical Python expression, the algorithm checks if there is a diff that matches any subexpression of it. If there is such a diff and this statement is an assignment, then the expression is evaluated in full, the environment is updated accordingly and a diff that targets the target of the assignment is created, in all other cases the statement is simply reevaluated.

6 RELATED WORK

Prior work related to ViDeTTe notebooks can be classified into the following categories:

Grammar-based visualizations. The concept of using a formal grammar to specify visualizations can be traced back to Wilkinson's "The Grammar of Graphics"[**Wilkinson:2005:GG:1088896**]. Since then multiple tools, such as Polaris [**stolte2002polaris**] (later commercialized as Tableau), ggplot2 [**wickham2009ggplot2**], Vega [**satyanarayan2015vega**, **vega**] and others [**ggvis**, **bostock2009protovis**, **d3**] have adopted this approach, thus enabling the description of visualizations (as well as the user interaction with such visualizations [**Wu:2016:DAI:2939502.2939517**, **satyanarayan2016reactive**]) in a terse and declarative manner. While the expressiveness of the employed grammars varies, their main focus is to effectively describe the specifics of the visualizations that will appear (for instance the dimensions of the chart, the colors, etc.), and not to describe data access or data processing and/or conversions. For that reason, such tools (due to their limited expressivity) do not provide constructs capable of retrieving data from diverse sources or performing arbitrary data transformations. Instead, the user has to provide imperative logic that performs these tasks manually and ultimately convert the dataset she wants to visualize into the format that is expected by the respective tool in order to generate the visualization.

Reactive Libraries. Some of the aforementioned tools (for instance ggplot2 [**wickham2009ggplot2**], Vega [**satyanarayan2015vega**, **vega**, **Altair**, **ipyvega**] and ggvis [**ggvis**]), do allow readers to interact with visualizations, however, none of these tools allow, data analysts to declaratively specify the wide range of side effects that ViDeTTe allows. Specifically, while these tools can create visualizations (and widgets) that depend on one another, they cannot declaratively describe more elaborate side-effects that include retrieving additional data, or issuing arbitrary computations (for instance, linear regression as was shown in this paper). They take as input the entire dataset that can be included in the produced visualizations throughout their life-cycle and they provide a declarative language with limited data processing (essentially describing a subset of the operations that can be expressed in SQL) and no data access capabilities. Additionally, these tools operate in isolation with respect to the rest of the notebook, which means that they cannot be used to generate co-dependent visualizations that appear across multiple notebook coding blocks. For those reasons they offer more limited data exploratory capabilities compared to ViDeTTe.

Data exploration. Tools that use the human factor as an integral part of data exploration have gained in popularity in the past years. Such tools provide a simple to use way that describes both computation and visualizations, thus providing the required toolkit to effectively explore data and steer computation. Some of these tools enable this process by providing a declarative language capable of performing such tasks (such as Zenvisage [**DBLP:journals/corr/SiddiquiKLKP16**] and Devise [**livny1997devise**]), while others (such as Vizdom [**crotty2015vizdom**], DataHub [**Krishnan:2016:AID:2994509.2994514**] and more[**derthick1997interactive**, **bhardwaj2015collaborative**, **zoumpatianos2015rinse**, **DBLP:conf/icde/LiarouI14**, **Kamat:2016:TSI:2939502.2939514**]) rely on the user interaction with a front-end, in order to compose the appropriate queries. As a result, the former tools assume a type of user with deep understanding in databases and query languages, while the latter offer a more user-friendly way of performing an analysis but at the same time allow a predetermined set of possible analyses. ViDeTTe notebooks bridge the gap between these two types of users; to a data analyst it provides constructs that allow rapid creation of notebooks for an arbitrary number of analyses, while to a non-technical user it provides the capability to interact with the produced visualization in order to further explore the underlying data.

7 CONCLUSION

We presented ViDeTTe an interactive notebook engine, that enhances interactive notebooks with capabilities that pertain to both data scientist that create notebooks and non-technical notebook readers. Specifically, ViDeTTe streamlines arduous tasks that usually have to be performed by data analysts, that compose such notebooks, while at the same time offering enhanced data exploratory capabilities to the readers of the published notebooks