Combining Design and Performance in a Data Visualization Management System

Eugene Wu, Fotis Psallidas, Zhengjie Miao, Haoci Zhang *,
Laura Rettig †, Yifan Wu ⋈, Thibault Sellam △
Columbia University, *Tsinghua University †University of Fribourg ⋈U.C. Berkeley △CWI
{ew2493, fp2273, zm2248}@columbia.edu zhanghaoci@gmail.com
laura.rettig@unifr.ch yifanwu@berkeley.edu thibault.sellam@cwi.nl

ABSTRACT

Interactive data visualizations have emerged as a prominent way to bring data exploration and analysis capabilities to both technical and non-technical users. Despite their ubiquity and importance across applications, multiple design- and performance-related challenges lurk beneath the *visualization creation process*. To meet these challenges, application designers either use visualization systems (e.g., Endeca, Tableau, and Splunk) that are tailored to domain-specific analyses, or manually design, implement, and optimize their own solutions. Unfortunately, both approaches typically slow down the creation process. In this paper, we describe the status of our progress towards an end-to-end relational approach in our data visualization management system (DVMS). We introduce DeVIL, a SQL-like language to express static as well as interactive visualizations as database views that combine user inputs modeled as event streams and database relations, and we show that DeVIL can express a range of interaction techniques across several taxonomies of interactions. We then describe how this relational lens enables a number of new functionalities and system design directions and highlight several of these directions. These include (a) the use of provenance queries to express and optimize interactions, (b) the application of concurrency control ideas to interactions, (c) a streaming framework to improve near-interactive visualizations, and (d) techniques to synthesize interactive interfaces tailored to end-users.

1. INTRODUCTION

The necessity for ease of exploration and analysis of big data volumes from both technical and non-technical users has turned interactive data visualizations to first class citizens in a wide variety of applications. Data exploration tools and decision-support systems [38, 50, 51], knowledge exploration interfaces [6, 54], machine learning and statistical suites [2, 3], or news media outlets [1] are just a few examples that expose end-users to interactive visualizations. This ubiquity of interactive visualizations highlights the importance of the visualization creation process. Yet this process is challenging with an ambitious goal set to fulfill the exploration and analysis needs of end-users. Consider the following example:

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

Figure 1: Example of an interactive visualization.

EXAMPLE 1 (REVENUE BREAKDOWN WITH CROSSFILTER). *Figure 1 visualizes the revenue breakdown of a company coupled with a crossfilter interaction technique [17]. Each individual chart performs a sum aggregation on revenue group by region (top left), by year (top right), by month (middle right), by day of week (bottom right), and by region (bottom left). For this example, we used the TPC-H benchmark. An interactive selection on the years (orange box at the top right) filters the other charts to the selected years 1997 and 1998. The green and gray portions of each bar correspond to the group-by sum aggregation for the non-filtered and filtered partitions, respectively.*

Creating the example interactive visualization relies on multiple areas of expertise. First, an expert must abstract out the domain-specific analyses and needs (e.g., the revenue breakdown is useful for decision-support). These analyses and needs are typically encoded as queries (e.g., group-by sum aggregations) or even hand-written programs. Then, a designer lifts these analyses into a more efficient visual domain by choosing or creating the appropriate visualizations and interactions (e.g., horizontal and vertical bar charts in a grid layout coupled with crossfilter interaction capabilities). Finally, a developer implements the design and optimizes the data layout and query or program execution to ensure that user interactions are sufficiently responsive. This process iterates, possibly many times, between design and prototyping until the desired design and performance characteristics are met.

There are two common ways to develop an interactive visualization: express it in a data visualization system or implement it by hand. Unfortunately, both approaches typically slow down the

iterative creation process. Data visualization systems are easy to use but limited to particular domain or type of interactions. For instance, Tableau [51] is an interactive interface to general OLAP queries, Splunk [50] is specialized for filter and aggregation operations over temporal log data, and crossfilter [17] uses linked selection interactions to identify correlated dimensions in a dataset. To overcome the limited functionality of off-the-shelf data visualization systems, application developers often implement the visualization by hand, and end up making ad-hoc design or performance-related choices that are heavily tailored to the specific interaction and design decisions in the visualization. These choices lead to a brittle interface that is difficult to modify without needing to rewrite much of the implementation. Furthermore, the prevalent use of event driven code makes interactive visualizations hard to maintain, extend, optimize, and reason about [61].

We believe that a declarative, relational approach towards building interactive visualizations can ease much of the developers burden. To this end, we are extending the Data Visualization Management System [20] (DVMS) to support composition of interactive visualizations and help insulate visualization developers from low-level design and optimization choices. There are a number of challenges that must be addressed:

Challenge 1: Language Design Consider implementing the interactive visualization of the above example under a client-system architecture. A manual implementation involves sophisticated server-side functionality to handle asynchronous requests from clients and client-side logic to handle user inputs by translating them to server requests (e.g, convert the selection of the year bars to a server request for crossfiltering). The client-side logic is then enhanced with logic for updates to the visualization state based on the server response. Furthermore, the developer would have to handle the communication between the two ends either using a response-request scheme (e.g., AJAX calls) or through persistent connections (e.g., WebSockets). Under both schemes there is a network latency during which the visualization state may be inconsistent. In response, the application developer would need to introduce consistency policies across a large set of programming abstractions to maintain consistent visualization state. This process highlights that the application developer is required to have a wide range of expertise to handle the multitude of programming abstractions involved in the overall system architecture. Instead, DVMS proposes a single language to the applications designers that is (a) expressive enough to support a wide variety of interactive visualizations, (b) easy enough to analyze and reason about to allow for automated optimizations and design choices, (c) familiar to developers to avoid time-consuming learning curves, and (d) coupled with debugging capabilities.

Challenge 2: Performance Optimization Prior research has found that increased latency has a detrimental effect on data exploration sessions [49], and modern visualization interfaces are designed to create the expectation of high interactivity. Unfortunately, although a number of optimization techniques have been developed in the visualization and database communities [22], optimizing performance continues to be a manual process. On one hand, existing visualization optimization techniques [31, 33, 34, 37, 47, 52] are specialized for specific scenarios that are neither applicable nor most suitable for every interactive visualization design. On the other hand, databases employ query optimizers that embody a wide range of optimizations [4, 8, 11, 18, 19, 29, 35, 40]. Yet they remain inaccessible because interactive visualizations are written imperatively, and it is unclear how different optimizations affect the end-to-end latency. Ultimately, each technique was developed in isolation, with its own set of performance and design assumptions, and the developer is left to manually combine these ideas together. In addition, the ability to use them from an end-to-end perspective relies on end-to-end information, which is difficult to analyze or not available in imperatively written systems. In this direction, a single data model can keep track of end-to-end information and can be flexible enough to decide between or combine multiple optimization techniques.

Challenge 3: Design A core use case for interactive visualizations is to abstract out patterns from existing analyses, encoded as e.g., python scripts or SQL queries, and lift them into a more efficient interactive domain [25]. In our example, the intended analyses is encoded as SQL queries that we mapped to charts in the visual space coupled with a crossfilter interaction. Even on the same data set, visualization developers need to prepare interfaces that adapt to a multitude of needs (some known, some unknown) and changing circumstances. In our example, the group-by day chart may not be important for decision-support, yet we included it in the interface. As a result, some current systems simply incorporate all common interactions into their interfaces [45], creating overcrowded and cluttered displays. To avoid cluttered displays, some visualization designers carry out user studies to decide the set of interactions to put on their interfaces. In this case, some interactions are often overlooked, as the scope of what analysts do with the data set is too large in big data scenarios. Even though both of these extremes are undesirable [11, 35], visualization designers are still using these methods as it is very hard to find the middle ground. To this end, a DVMS can keep track of users' interaction history and propose design choices to application developers based on them.

In the spirit of prior relational approaches towards web development [21, 36] and interactive data cleaning [42], our primary hypothesis is that a single, well-understood abstraction—relational languages—can be suitable for both expressing the entire visualization application, and extending the application with richer design, development, and optimization support. In the following sections, we present our data model and DeVIL, our SQL-like language, for static and interactive visualization and our DVMS system architecture (Section 2). Then, we discuss our recent research efforts for a DVMS ecosystem (Section 3), which is a set of DVMS extensions to facilitate the visualization creation process. In particular, we introduce provenance for visualization interactions to provide novel functionality and optimizations (Section 3.1), introduce distributed consistency ideas to visualization interactions (Section 3.2), improve nearly-interactive visualizations (Section 3.3), and synthesize novel interactive interfaces tailored to the user (Section 3.4).

2. OVERVIEW

In this section we provide an overview of our end-to-end relational approach for expressing interactive visualizations, as well as a description of the current system. The direct benefit of our relational approach is that we can bring traditional, well-founded optimizations from the database literature (e.g., query optimization, physical database tuning, and view maintenance) to the visualization domain. In addition, this approach allows us to push down further optimization and design choices into the DVMS, by analyzing familiar SQL-like statements and corresponding workflows.

To better illustrate key points in our discussion we use a simple linked brushing [10] example that most visualization toolkits and systems implement imperatively:

EXAMPLE 2 (LINKED BRUSHING). *Figure 2 visualizes sales data and shows step-by-step an application of linked brushing: initially, a static visualization is composed of a scatterplot that correlates the sales revenue and profit of each product, and a*

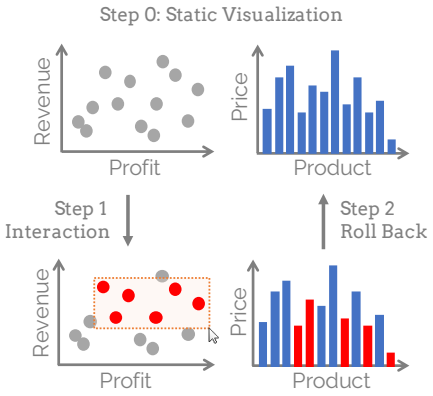


Figure 2: Brushing and Linking example.

histogram that shows the stock price again for each product. (1) shows a mouse drag interaction that selects a rectangular region in the scatterplot, alongside all marks (i.e., circles) inside the selection. The subset of products that corresponds to the highlighted circles are "selected" by turning red, as are the histogram bars corresponding to the selected products. After finishing the selection, the user may deselect and reset the visualization (2), keep the points highlighted, or perform another selection.

2.1 Data Model and Language Overview

In this section, we will first "relationalize" static visualizations. Then we introduce user interactions as event streams that we subsequently use to express interactive visualizations as database views involving joins between event streams and static visualizations. We will first assume a single user, single interaction scenario, and then discuss our current approach to support multiple interactions.

2.1.1 Static Visualizations

In line with prior work in the visualization community [46, 56], we model a static visualization as a mapping from a set of database relations $\mathcal{R} = \{R_1, \ldots, R_{|\mathcal{R}|}\}$ in the data domain to relations in the visual domain. We model the visual domain using two types of relations: `Marks` relations that describe the circles, polygons, and other shapes to be rendered in the visualization as well as a `Pixels` type of relation that models the rasterized pixels shown to the user. Each relation of the former type corresponds to a specific mark type (e.g., line, circle, or rectangle), with attributes including the geometry and visual encoding of the corresponding marks. The later type is a special pixels relation `P(x, y, RGBA)` that models the color and transparency encoding at every pixel coordinate. However, its contents are not materialized; rather, they are maintained by the rendering device. Modeling it as a relation, however, is useful for analysis purposes, as we elaborate on in Section 3.2.

The mapping of the static visualization encapsulates the data transformations (e.g., aggregation and scaling) that encode data summaries as geometry and visual encodings (e.g., height and color of a circle). In our work, the data is represented under the relational model, and the mapping is expressed using relational queries.

For example, DeVIL 1 shows a query that maps sales data in the relation `Sales(productId, price, profit, revenue, productName)` to a marks relation `SPLOT_POINTS` that represents the circles of the scatterplot in Figure 2. Each projection clause defines an attribute of the circle mark, such as the `radius` as well as `stroke` and `fill` colors. The `linear_scale` UDFs linearly transform the sales revenue and profit to their corresponding pixel coordinates—the UDF uses the `scale_x` and `scale_y`

```
SPLOT_POINTS =
SELECT
 8 AS radius,
 'gray' AS stroke,
 'gray' AS fill,
 linear_scale(Sales.revenue, scale_x) AS center_x,
 linear_scale(Sales.profit, scale_y) AS center_y,
 productId
FROM Sales, scale_x, scale_y;
P = render(SELECT * FROM SPLOT_POINTS);
```

DeVIL 1: Static visualization for the scatterplot of our example.

relations, which include the minimum and maximum values of the revenue and profit attributes, to compute the transformation. The last attribute, namely, `productId`, is typically used in visualizations as a way to ensure a correspondence between the rendered mark and the input record—Section 3.1 will describe provenance extensions that enable this correspondence in a declarative manner. Finally, marks relations are rendered using the `render` table UDF. Similar selection queries and render functions can be used to define the static visualizations of the histogram and axes in Figure 2.

DeVIL supports the specification and usage of record and table user-defined functions. However, they are restricted to pure functions without side-effects. The only exception is for rendering functions, that may produce visual side effects—they can only be executed on marks relations, and logically produce the pixels table.

The above reviews the procedures described in [20] and illustrates how static visualizations can be expressed as database views. Next, we describe our extensions towards interactive visualizations.

2.1.2 Interactive Visualizations

DeVIL models interactions as the effects of user interactions on database views. To do so, we capture user-generated low-level events (e.g., mouse down, move, and up) as event streams, and extract compound events (e.g., mouse drag) as patterns over the streams of low-level events. Then, we introduce interactive visualizations as database views involving event streams. This model allows us to draw a direct analogy between an interaction and a database transaction in that each compound event, that defines an interaction, may either transition the database to a new version or rollback to the version right before the beginning of an interaction.

Next, we introduce streams of user-generated low-level and compound events in our data model and express interactive visualizations as database views that involve event streams, visual primitives, and base data. Then, we provide a technical definition of an interaction and provide constructs for composition of interactions.

Events When a user performs an interaction such as dragging a mouse, it is interpreted by most systems (e.g., browsers, GUI frameworks) as a sequence of low-level events such as mouse down, or mouse move. For this reason, we model user interactions as event streams of low-level events, and high-level interactions (e.g., mouse drag) as compound events atop streams of low-level events.

To capture event streams of low-level events, we adopt the data model of CQL [7]: given an alphabet of low-level events with predefined schemas Σ (e.g., Σ = {mouse down, key press, ...}), we model a stream of low-level events as an (unbounded) set of ordered pairs $\langle s, t \rangle$, where $s \in \Sigma$ and t is the time when a user performed s.

To extract compound events from low-level event streams we could leverage a number of automata-based approaches that identify complex patterns in event streams [7, 13, 28, 30, 47, 58]; regular expression-based languages such as Proton [30] are used in the user interface literature to recognize user gestures. We borrow these ideas in a sequence matching language similar to SASE [58], which compiles into a nondeterministic finite automaton (NFA).

```
C =
EVENT MOUSE_DOWN AS D, MOUSE_MOVE* AS M*, MOUSE_UP AS U
WHERE FORALL m IN M m.y > 5
RETURN
   (D.t, D.x, D.y, 0 AS dx, 0 AS dy),
   (M.t, D.x, D.y, (M.x - D.x) AS dx, (M.y - D.y) AS dy)
```

DeVIL 2: Event statement to generate a compound event stream.

DeVIL 2 is an example event statement that defines a compound event stream C as a sequence of mouse down, repeated mouse move, and mouse up events; we allow kleene closures to express repeated events. Non-matching event types (e.g., a key press) as well as events that fail predicates in the WHERE clause are filtered from the input stream and not processed by the NFA. For instance, (e.g., D.y > 20 removes mouse down events below 20 pixels from the input stream. The only exceptions are existential and universal quantifiers, such as the FORALL predicate in the example, which trigger a reject state upon failure. Finally, the RETURN clause defines a sequence of union-compatible projection statements, and concatenates all statements that can be evaluated by the matching events. For instance, DeVIL 2 first emits the mouse down event, followed by each move event along with its distance from the down event.

t	x	y	dx	dy	Input event
0	5	15	0	0	MOUSE_DOWN(0,5,15)
1	5	15	1	2	MOUSE_MOVE(1,6,17)
			...more MOUSE_MOVE events ...		
40	5	15	5	-5	MOUSE_MOVE(40,10,10)
			MOUSE_UP(41,10,10) terminates the query		

Table 1: Contents of the event table C in our example after a potential sequence of user-triggered low-level events.

As a concrete example, Table 1 illustrates the state of the relation C during a user's drag movement. The mouse down at t=0 inserts the first record, based on the first projection statement of the RETURN clause. Note, no record is inserted in C, at t=0, due to the second projection statement because it involves mouse move events that have not happened yet. Subsequent mouse move events insert corresponding records into C based on the second projection statement of the RETURN clause. For every mouse move, no record is inserted in C due to the first projection statement because it does not involve mouse move events. Finally, the mouse up event transitions the NFA to an accept state and terminates insertions into the relation. Note, no record is inserted in C due to the mouse up event because it is not involved in any projection statement.

As we will demonstrate next, an interactive visualization can be expressed as a view over the complex event table C and the database relations and views. Thus, insertions into C will trigger view updates that change the visualization in response to user interactions. The EVENT statement defines the boundaries of an atomic user interaction, and we currently use it to define transaction boundaries—the start and accept states correspond to the beginning and end of the transaction, and the reject state corresponds to an abort.

The key difference from traditional transactions is that the "uncommitted" state—in this case, the state of the visualization throughout the mouse move events—is exposed to the user in the form of visualization updates. In the current system, abort is equivalent to clearing the compound event table C in order to roll back to the state of the visualization before the interaction. Finally, to prevent never-ending transactions, we constrain statements to sequences that end with a non-repeating event. As a result, the underlying NFA can transition only once to an accept state that ends the transaction and commits by persisting the new visualization state.

Putting it Together We now describe how to express the brushing interaction in Figure 2 as a view over database relations and user event streams.

```
selected     = SELECT SP.productId
               FROM   C, SPLOT_POINTS@vnow-1 AS SP
               WHERE  in_rectangle(bbox(C), SP);

SPLOT_POINTS = SELECT ..., 'gray' AS fill
               FROM Sales, scale_x, scale_y
               WHERE productId NOT IN selected
               UNION
               SELECT ..., 'red' AS fill
               FROM Sales, scale_x, scale_y
               WHERE productId IN selected;
```

DeVIL 3: Selection interaction for the example scatterplot.

To start off, we may specify the set of selected marks using a join between C and the scatterplot marks relation SPLOT_POINTS, as shown above in DeVIL 3: bbox is shorthand for a query that computes the selection box of the mouse drag events in C, and in_rectangle is shorthand for a predicate that checks whether a mark SP intersects with the selection box.

As the event query populates C, the selected relation will update accordingly. By redefining the scatterplot marks relation to perform different projections for selected and non-selected records, we are able to express brushing. Similarly, we can update the contents of the histogram to highlight bars related to the selected records. Because both views coordinate on the selected view we have the desired effect of linked brushing for our example.

The main challenge in expressing this interaction is that these statements introduce recursion—the selected view depends on the SPLOT_POINTS view and vice versa. To address these issues, DeVIL disallows recursive statements by allowing the developer to specify past versions of relations. Specifically, developers can specify the committed state of a relation i transactions ago by adding the suffix @{vnow-i} to a relation name; the syntax @{tnow-j} specifies the state of a relation j events ago within the current transaction. This versioning approach highlights the relationships between our notions of transactions and interaction events, and simplifies support for interactions such as undo or mouse trails. For example, DeVIL 3 computes the selected marks by performing hit testing on the marks SPLOT_POINTS@{vnow-1}, which is the version of SPLOT_POINTS at the beginning of the current interaction as expressed by the event statement in DeVIL 2.

What is an Interaction? The interactive visualization in DeVIL 3 provides us with a better understanding of what we consider as an interaction in DeVIL. More specifically, we technically define an interaction as an object that encapsulates an event stream along with the view statements that involve the event stream. With this abstraction in mind, we can better analyze interactions and introduce critical operations of the visualization design process (e.g., add or remove interactions to or from static visualizations). For instance, adding a single interaction to a static visualization results in introducing an event stream along with the rewrite of the corresponding view statement, as we showed in DeVIL 3. Beyond expressing a single interaction, composition of interactions is an important task for the visualization domain, that we discuss next.

Composition of Interactions DeVIL can combine multiple interactions as sequences to support multi-step operations, or as interleaved in parallel to support simultaneous inputs such as the mouse and keyboard. One difficulty is to define how the two interactions share state. In DeVIL, an interaction is defined by an event stream as well as the views that use the stream, as we discussed above. Now, suppose the developer defines I1 as a brushing in-

teraction and `I2` as an interaction to drag a set of marks. Suppose the designer wants to express `I1 + I2` as a brushing action to select marks followed by a drag action to move the set of selected marks—`I2` will need information about the marks selected by `I1`. Although it is well known how to combine two NFAs (i.e., for introduction of a single event stream for both interactions), it is unclear how to unambiguously combine the view statements of the two interactions. We currently rely on the developer to define a merging function $\texttt{merge}(I_1, I_2) \rightarrow I_{\texttt{combined}}$ that transforms a sequence of two interactions, I_1 followed by I_2, into a combined interaction. The function can re-write a subset of the statements in I_2 and has read-only access to the relations in I_1.

The second difficulty is reasoning about conflicts, ambiguities, or unintended effects that can arise from interaction composition. Although some recent debugging tools [15, 27] visualize the internal state, manually analyzing the internal state of complex interfaces is tedious and low-level. There is a need for tools that can warn developers of potential interaction-related errors, or illustrate possible interpretations of a set of interaction statements. Towards this direction, we are currently studying the extent that two DeVIL interaction statements can be statically analyzed and prompt the application designers for potential erroneous behavior—hence, expediting the time-consuming debugging process. For example, we can disambiguate interaction statements by changing the event statement definitions [30]; partitioning when the event statements are evaluated by time (state of the visualization) or space (type of widgets or marks); or assigning interaction priorities. Resolution rules for one combination of interactions may be templated and applied to others with similar semantics.

So far we have laid out the fundamental concepts of DeVIL. Next, using only these concepts, we evaluate the expressivity of DeVIL against taxonomies of interactions.

2.1.3 *Expressivity and Applications of DeVIL*

We have found that the primary classes of interaction techniques—interactive selection, changing visual encodings, adding or removing marks, coordinated views, and undo/redo—that are common across several interaction taxonomies [57, 63] can be readily expressed in DeVIL based on the constructs presented above.

Interactive selection allows users to point to and select objects in the visualization, and is a building block for more complex visual interactions [37, 46, 57]; it was expressed above as a join between the user's interaction event stream and the rendered marks relations. Visual encodings can change how input data is visualized (e.g., by color, by position, or by different types of marks) and naturally translates into modifications of a projection clause (e.g., change from gray to red color in DeVIL 3). Adding or removing marks is natively supported by inserting or removing data in the underlying database relations and performing view updates, or by manipulating selection predicates. Coordinated views, such as the multi-view selection in Figure 2, can be expressed by sharing relations between multiple marks relation definitions, as we showed with the `selected` view. Finally, undo and redo is supported by the versioning semantics within and across interactions.

We have shown that a wide range of interactive visualizations can be expressed using well understood constructs from the relational world. This relational lens enables us to directly borrow database concepts such as query optimization, physical database tuning, or view maintenance to the interactive visualization domain. In the subsequent sections, we will use this lens to 1) introduce provenance to visualization interactions in order to provide novel functionality and optimizations (Section 3.1), 2) apply and extend distributed consistency ideas to visualization interactions (Section 3.2), 3) improve nearly-interactive visualizations (Section 3.3), and 4) synthesize novel interactive interfaces that are tailored to the user (Section 3.4).

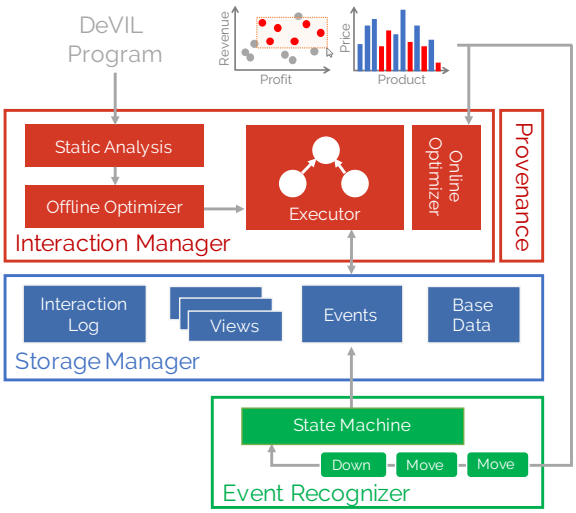


Figure 3: DVMS System Architecture.

2.2 DVMS Architecture

Figure 3 shows the DVMS architecture. The *Interaction Management (IM) Engine* translates DeVIL programs into a *visualization workflow*: Each DeVIL statement is an assignment statement, where the right hand side defines an operator whose output is the left hand side of the assignment statement; relations and views involved in the right hand side are treated as inputs to the operator. Then, IM performs static analysis (e.g., detecting ambiguity between interactions), offline optimization (e.g., rule-based optimization, and physical database planning), and generates appropriate view definitions based on the visualization workflow. These views are evaluated by the *Executor* which ultimately outputs the pixels table P that is perceived by the user. The *Event Recognizer* compiles the event statements in the DeVIL program into state machines, and matches them against the stream of user input events. Matches are inserted into the corresponding *Events* tables in the *Storage Manager*. In addition, the Storage Manager stores the base relations, intermediate results, and the materialized views and their definitions. The inserts trigger the *Executor* to recompute the view definitions, which ultimately updates the marks and pixels tables.

The *Online Optimizer* captures event and visualization state information throughout the user's interactions and dynamically re-optimizes the visualization. For instance, Sections 3.3 describe how we can leverage performance and online prediction models to re-optimize the interactions and visualization design. Finally, the *Provenance* component spans the three major components, and provides lineage and logging support to enable further interaction optimizations and functionality, as we elaborate on in Section 3.1.

Our current implementation uses a modified SQLite3 parser to parse DeVIL programs, and a ECMAScript-compiled instance of SQLite3 to store the base data and compute the mark tables. A table UDF computes layouts, while both row and table render UDFs map the mark tables to DOM SVG and canvas elements.

3. DVMS ECOSYSTEM

In this section, we present our DVMS ecosystem—a set of extensions to facilitate the interactive visualization creation process. Many of these extensions are in their early stages and we describe our current approaches and findings.

3.1 Provenance for Visualizations

A DeVIL program is translated into a visualization worfklow. Provenance operations over visualization workflows can trace between the inputs (e.g., data items, marks of former visualization states, or events) that generated elements of the output (e.g., marks of the current visualization state). Our main observation is that provenance (more specifically, lineage) operations can serve as primitives for a wide range of important functionality in the visualization domain including linking and coordination of views, interactive debugging of applications [27], deconstruction and restyling of visualizations [23], data explanations [44, 59], or provenance visualization [9, 26, 41].

Linking and Coordination In our example of scatterplot specification in DeVIL 1, we noted that `productId` is an annotation connecting output circles of the scatterplot to the corresponding product. Then, we used the `productId` to specify the `selected` view and we introduced the interactive versions of both the scatterplot and the histogram based on the `selected` view.

The shortcomings of this approach are two-fold. First, this approach breaks our data model because developers must manually augment their queries with provenance annotations; if these queries change, then the annotation strategy must also change. Second, annotations can introduce significant overhead and the space of optimization strategies for a given visualization workflow is infeasible for an end-developer to manually explore. For instance, in the example above we could have annotated the marks of the scatterplot and histogram with mark ids and materialized the correspondences between scatterplot points and histogram bars in a separate table (i.e., `circleId` → `barId`). If the two plots visualize subsets of products the developer should also consider ways to keep the mapping only for the intersection of the subsets. However, increasing the number of linked views increases the number of correspondences quadratically and this strategy quickly becomes unwieldy.

Our primary observation is that the above annotations are simply manual instrumentations of the workflow to express a backward query from the selected marks to the input records, and a forward query from those records to the histogram bars. We have extended DeVIL to support backward and forward trace statements. Following is an example of how we can accomplish our brushed linking example using a backward trace operator:

```
B             = BACKWARD TRACE
                FROM SPLOT_POINTS@vnow-1 AS SP, C
                WHERE in_rectangle(bbox(C), SP)
                TO Sales;

                          ▷ SPLOT_POINTS without productId
SPLOT_POINTS = SELECT ..., 'red' AS fill
               FROM B
               UNION
               SELECT ..., 'gray' AS fill
               FROM (Sales MINUS B)

HIST         = SELECT ..., 'red' AS fill
               FROM B
               UNION
               SELECT ..., 'blue' AS fill
               FROM (Sales MINUS B)
```

DeVIL 4: Linked brushing using provenance operations.

The `BACKWARD TRACE` statement above resembles the structure of a `SELECT` query. More specifically, the `FROM` clause along with the `WHERE` clause denote a join among the event stream `C` and the scatterplot that determines the selected circles. The `TO` clause denotes the relation to trace backwards the result of the join—in this case the `Sales` relation. The interactive histogram and scatterplot are defined based on the partition {`Sales`\`B`, `B`}: circles and bars for the backward traced subset `B` are colored red while the unselected marks in `Sales`\`B` are colored gray and blue, respectively. Hence, events on the stream `C` change the backward traced subset that, in turn, changes the histogram and the scatterplot with the desired linked brushing effect.

Beyond linking and coordination, provenance can support a large range of functionality including:

Interaction Debugging: provenance is traditionally used for workflow debugging. For the visualization domain, there are three main debugging operations [27], all enabled through provenance support. First, provenance can expose the state of the visualization workflow (i.e., data, marks, pixels, and events) for inspection. Furthermore, provenance can identify input-output dependencies between operators of the workflow. Finally, provenance along with the temporal semantics of DeVIL (i.e., `@vnow-k`) lets developers inspect and debug previous states of the workflow through the versioning mechanism.

Deconstruction and Restyling: Harper et al. [23] present a technique to extract data from marks in a D3 visualization and revisualize the data using new visual enocdings. Their technique relied on D3, which annotates each mark with the full data object that was used to generate the mark. Native provenance support can support such restyling techniques out of the box.

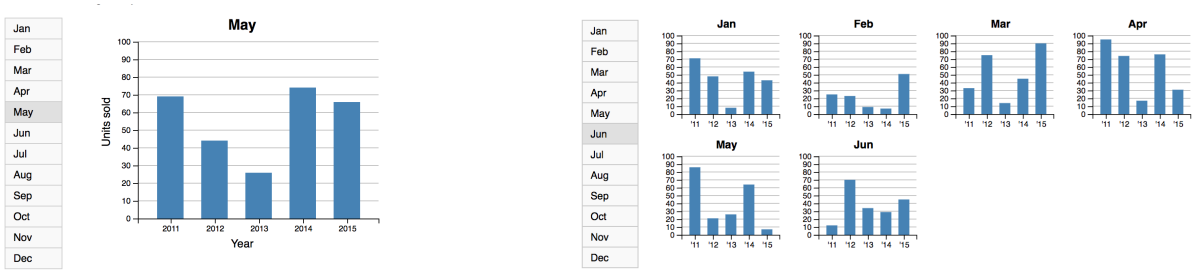
Visualization Explanation: outlier explanation techniques [44, 59] rely on the ability to trace outliers in the visualization back to their inputs in order to identify the inputs to the explanation generation algorithms. Thus provenance support can allow such techniques to be modeled as a user defined function and be enabled for any DeVIL visualization. Further, the explanations themselves may also be visualizations using additional DeVIL statements.

Provenance Visualization: the problem of provenance visualization [9, 26, 41] is very challenging; partly because it quickly devolves into a network visualization problem. DeVIL presents an interesting special case where the provenance generated by the visualization workflow can use the DeVIL specification itself as hints for how to render the provenance data.

We believe the connection between provenance and interactions offers insights in both areas. Typical lineage systems materialize and index the system's lineage information to speed up lineage queries, but materialization and indexing costs can be substantial [60]. In addition, the amount of provenance data can easily be so large that the cost of querying it is substantial. Despite these limitations, it is clear that important classes of provenance queries can be optimized to respond in interactive latencies—we have seen that linked brushing can be expressed as provenance statements and numerous existing systems [17, 31, 33, 39] have scaled interactive linked brushing to large datasets. We believe this is possible because most applications such as visualizations do not use lineage data per se, but feed it as input to aggregation or filtering queries (e.g., DeVIL 4). This lets the provenance subsystem avoid unnecessarily materializing lineage data that will not be used. We are studying how this observation can be combined with physical database design methods to optimize both provenance workloads as well as interactive visualization workflows.

3.2 Concurrent and Consistent Interactions

Interactive visualizations are increasingly expected to operate in environments where the amount of time needed to respond to user inputs is unpredictable. This may be due to a visual interface that issues data processing tasks over an unpredictable network such as the internet, or due to multi-tenancy issues such as concurrent use of a backend data processing system.



(a) **(b)**

Figure 4: (a) The user hovers over a month to update the bar-chart. (b) A MVCC design: when users issue concurrent requests, a new copy of the chart is created for each request, resulting in the small multiples visualization.

Traditionally, when datasets small enough to be loaded locally and processed nearly instantaneously, interactive visualizations can simply write synchronous code to handle data processing. However, the advent of "big data" and client server visualization architectures have made it difficult, if not impossible, to guarantee sub $100ms$ responses times. When user inputs cannot be processed quickly enough, visualization systems either block user inputs or the interface until the processing is completed (e.g., a wait icon), or allow the processing to occur asynchronously from further user inputs. As described in prior work [61], the former can drastically degrade the user experience, while the latter is susceptible to logical errors caused by arbitrary interleavings of the user requests that are notoriously challenging debug and program correctly.

The database community is currently developing novel techniques to reduce data processing times to sufficiently low levels that can support even interfaces that block user input [8, 18, 31]. In other words, these approaches are focused on one design criteria—query latency—as the means to improve usability. Given the user-facing nature of interactive visualizations, we hypothesize a broader design space that can be optimized. For example, users performing tasks that do not require order, such as finding if a series of data pass a threshold condition, might be more resilient to asynchronous updates. Another hypothesis in the design space is cumulative visualizations could alleviate the need for order and memory in many asynchronous visualizations, such as non-displacing visualization of data onto different regions on a map.

In order to answer these questions, we believe it is important to understand the user-level constraints that help direct system level optimizations. Consider that a single study [32], which found that an additional delay of $500ms$ negatively impacts visual exploration, has served as a guiding post for many visualization optimization research, mentioned previously. To this end, we have designed a series of user studies to understand the design constraints around asynchrony in interactive visualizations; the results will be used to develop a declarative asynchrony management component for interactive visualization systems.

These studies are important because every visualization system currently makes implicit assumptions about how asynchrony is handled. A system may choose to disallow asynchrony by blocking user input, require that the user never sees outdated requests, or simply enable asynchrony without any form of coordination (e.g., vanilla AJAX-based systems). In each case, it is not clear whether the choice is too conservative and leave performance on the table, or simply incorrect because it can cause the user to be confused or reach incorrect conclusions.

One benefit of DeVIL's relation-based model is that it is natural to borrow ideas from classic database concurrency control. For instance, blocking user input can be viewed as serially ordering user inputs by locking the entire interface (e.g., pixels table in DeVIL) until the pixel table has been updated. Turning off concurrency control is equivalent to vanilla AJAX-based asynchrony. Similarly, locking, OCC, eventual consistency, and other concurrency control mechanisms may be applied to DeVIL programs.

In this light, we hypothesize that the classic database notion of serializability is too stringent in the context of interactive visualizations, and that it is possible to change what "serializability" means by studying how different types of reorderings affect the user's actions and conclusions. Understanding the user-level constraints of reordering could in turn promote UI designs that are more resilient to unpredictable time to response, imposing a less stringent requirement on data processing times to enable usable interactions.

Our baseline user study shows participants a simple interactive visualization consisting of a single interaction widget (e.g., slider) that changes the contents of a single visualization (e.g., bar chart). We study how two orthogonal dimensions affect the speed and accuracy that participants complete a range of judgment tasks. The judgment tasks are designed to vary in difficulty from identifying whether a target bar ever exceeds a threshold value to identifying a trend over time. Some tasks, such as the threshold task, are friendly to asynchrony because the order that the visualization updates due to user interactions does not affect the answer, while other tasks, such as trend estimation, requires that the order of user inputs is reflected in the order of visualization updates.

In our experiments we vary the mean request latency to understand how user interactions change as response latency increases. We have independently two classes of policies. Reordering (concurrency control) policies vary how user input are handled and include no order constraints(`No CC`); fully serializing responses (`Serial`); enforcing responses to be in-order by discarding out-of-order responses (`Discard`); or only rendering the result of the most recent interaction and discarding the rest (`Most Recent`). We also study visual design policies—the one we study is a form of "multi-visual concurrency control" (`MVCC`), which replicates the bar chart for each inflight request so that their updates to the pixels do not conflict. As an example of `MVCC`, consider Figure 4.a: when the user hovers of the month facets, the single bar chart is updated directly. In contrast, a `MVCC` variant will create a *copy* of the bar chart (e.g., a new "version") for each month that the user hovers over, as shown in Figure 4.b.

Figure 5 shows the average completion time for the threshold task under the above policies and visualization. Combined with measures of users' interaction behavior, the results suggest that concurrency-friendly policies allow users to generate more and make use of concurrent requests and subsequently reduce the task completion times, as compared to concurrency-unfriendly policies. Although each of the above policies have little difference when there

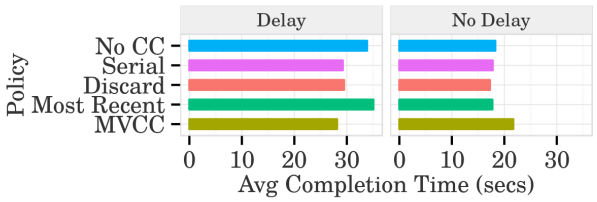


Figure 5: Comparison of average completion time of threshold task by policy, under different delay conditions.

is no response delay (in fact, `MVCC` is slightly slower), there is a clear difference when random delay (mean=2.5sec) is introduced. `No CC` and `Most Recent` take the most time to complete. `Most Recent` only shows the most recent user input (e.g., if the user drags a slider, she only sees the data at the position the slider stops). In both cases, users appear to serialize their own input—by hovering over a facet, waiting to see the visualization update, and then performing the next interaction—in order to understand the changes reflected in the chart. In contrast, we hypothesize that `Serial` and `Discard` improves completion times because the order that the visualization updates remains in the same order as the user inputs (though `Discard` may drop some updates). Finally, `MVCC` has the lowest completion time; from analyzing user event logs, we find that users hover over a large number of facets to issue many requests, and wait for multiple visualizations to appear. We have run this experiment on a perceptually more difficult judgment task and found these effects to be more pronounced.

The above are promising results from a preliminary study. We are currently performing a larger scale study under a wider range of tasks, latency profiles, reordering, and design policies. The longer term goal is to identify a set of design principles for asynchronous visualization interactions, and to ultimately extend DeVIL with asynchrony constraints that specify how new user events trigger view updates to the marks and pixels relations.

3.3 Improving Near-interactive Visualizations

Modern interactive visualization systems [51, 55] typically consist of a client and server (perhaps on the same machine), and painstakingly optimize the performance of their interfaces—by building auxiliary data structures, pre-computing large portions of the visualization state, pre-computing samples, or materializing intermediate data—all in the effort to reduce the latency to respond to user inputs. In most situations, the applications are able to support sub-second latencies that are nearly interactive (150 – 700ms), but still slower than the < 100ms threshold for interactivity. For instance, Tableau Web [53] heavily optimizes their client including the use of HTML5 canvas, while other visualization systems precompute every possible view [14].

We believe a major contributing factor to the near interactive latency is the request-response model of interaction, where the system waits for a user to perform an interaction, which triggers a query and render request that the system processes. Waiting until the user performs the request means that even minor variations in processing or network transfer times can affect usability. Although the previous section described our approaches to support concurrency as a response to latency, we are also developing techniques to push these near-interactive visualizations past this final threshold.

To this end, we are exploring a continuously streaming framework between the client and server that is composed of two characteristics. First, given a bandwidth bound, the server continuously sends data at bandwidth capacity to the client; the contents of the data stream are based on a shared speculative model of the user's intents [18]that the client continuously sends to the server. This user intent model estimates the probability $P(a_i, t) \in [0, 1]$ that the user will perform action a_i within time t. Second, the data that is sent to the client is progressively encoded in a way so that the client can, at any time, render the partial set of data it has received from the server.

This framework is well suited for settings such as interactive visualizations, where an offline phase has pre-generated data structures such as data cubes or materialized views. For instance, many modern visualization systems pre-compute datacubes and store them as individual slices of data tiles [8, 33]. In such a setting, the client typically needs to request a tile and fully download the tile before it can fully render the contents in the visualization. However, data tiles can readily be progressively encoded, say, by using wavelet compression. Progressively encoded offline data structures allows the server, when the user intent model is relatively uniform, to interleave data relevant to a large range of future actions so that *some visualization* is always available.

We find that this framework is similar to the partial task scheduling problem addressed by He et al. [24]. In their work, tasks are annotated with hard deadlines, and the schedule may partially execute a task; the utility of a partially executed task is modeled by a concave function. He et al. find that the problem of maximizing the total expected utility can be modeled as a convex optimization problem. We can translate our streaming problem into one where each task is to fetch a data tile, the deadline is 100ms from the time that the user requests the tile, and the utility is defined by the particular progressive encoding technique.

He et al. [24] assume that task deadlines are fixed and tasks can be discarded once the deadline has passed. In contrast, we only have access to the user intent model, which continuously changes in response to user actions. To address the lack of a well defined deadline, we scale the utility function for a given data tile in proportion to its likelihood. In addition, if the deadline for a tile passes, we simply reschedule it the next time the scheduler is executed. Finally, to address the continuous changes in the user intent model, we periodically re-run the scheduler (in our case, every 50ms), and are looking into more incremental approaches that do not require rerunning the scheduling algorithm for minor changes in the model predictions.

The above approach is predicated the accuracy of a user intent model. Past works focus on predicting the logical action—roll-up, drill-down, pan, zoom—in the form of a SQL query that the user will perform. However, such predictive models encounter sparsity issues due to the large number of possible operations that a modern visualization interface will provide; thus there is an inherent tension between predictive accuracy and the scope of the operations that can be predicted. Our main observation is that user express their interactions using a constrained input modality (typically a mouse) for which simple models work very well. Because DeVIL preserves the logical mapping from pixel locations to the interactions that can be expressed, it allows us to develop a highly accurate predictive model that is well suited for near-interactive visualizations. Using off the shelf models [12], the model is 82% accurate at predicting the widget that the user will interact with in 200ms. This allows the server to accurately speculate the user's intended requests and fetch partial results for a large fraction of them. A secondary benefit is that the predictive model can be trained from any type of mouse trace data, and is not dependent on data collected for a specific visualization.

3.4 Precision Interfaces

Previously, we discussed *how* to create interactive visualizations. Now, we present methods to help developers decide *what* to build. Automating visualization design is important for two reasons:

First, the "one size fits all" design approach that creates a single visual analytic interface for all users is suboptimal. Different users are interested in different types of exploration and analysis. But user requirements are a moving target: they are often fuzzy, variable and unpredictable. Creating tailor-made interfaces from scratch can turn out to be a slow, iterative design process. One way to solve this problem is to let users compose their own interfaces, with tools such as Mac Automator, Scratch [43] or Sikuli [62]. But this solution requires patience and manual effort even from end-users without technical expertise.

Second, automating design decisions simplifies software development and empowers non-technical users. Many professionals manipulate databases on a daily basis, but have neither the knowledge nor the interest to write applications. For most scientists; analysts; business consultants; and journalists, good practices of software engineering is all but a priority. Those users rely on IT services to generate "dashboards" or run queries on their behalf. For instance, financial institutions hire consultants and tactical developers to build ad-hoc "reporting applications." Yet, this process is costly, it creates heavy strategic dependencies, and it increases the exposure of potentially sensitive data.

How can we automate interface design? A promising direction is to *exploit user interaction logs*. Rather than running extensive user studies or creating overly cluttered interfaces, DVMS can infer the goals of end-users directly from the queries or scripts that they have submitted to the system. Furthermore, many analyses involve modifying parameters or scripts in structured, incremental ways, rather than complete program rewrites. For instance, a user will repeatedly tune filtering or grouping clauses in a SQL query before tweaking the query in another way to study another facet of the data. To illustrate, we examined a sample of 125,600 SQL queries from the log of Sloan Digital Sky Survey (SDSS) [5, 48], taken between November 28th and 30th, 2004. We managed to map more than 99.1% of those statements to only 6 query templates. We suggest to exploit this structure: for each possible "tweak" of the query templates, we can create an interaction on a custom interface.

A key challenge is that analysts have their own preferred language (e.g., SQL or python), and developing automated ways of detecting and mapping syntactic changes in program code for every language to interaction semantics is infeasible. Our primary observation is that all programs are parsed into abstract syntax trees (ASTs) before execution, and that tweaks and incremental program changes amount to subtree differences at the AST level. Thus, an AST-based approach can generalize to nearly any language.

Even for the same language, the AST structure of the same, say, SQL query can be different when parsed by different parsers. In addition, since each widget corresponds to one or a few fixed patterns of sub-tree differences, these patterns will also be depending on the specific parser. Therefore, although the number of possible widgets is limited, we are still unable to automatically mine fixed patterns of tree differences and map them to the set of predefined interactions and widgets. Instead, we assume that a developer with a basic understanding of the language parser can express a set of mappings in a simple SQL-like [16] language to express AST differences that are interesting (e.g., a change of a numeric parameter of a function). We then compare each pair of queries in the query log, and map the queries that satisfy the predefined rules to the set of interactions (e.g., to a slider) as stated in the program. The following is an example of a program written in the language:

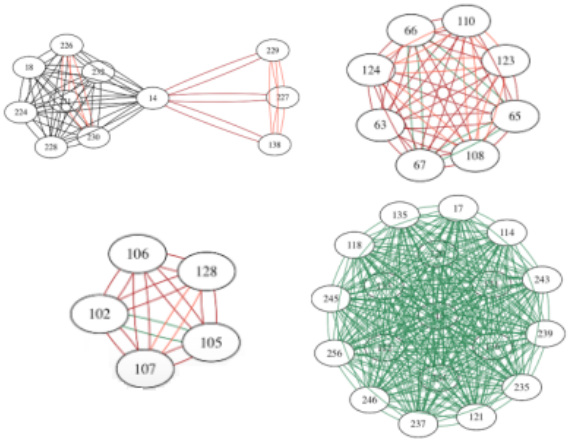


Figure 6: Interaction graphs, derived from the query log of the SDSS Skyserver database (November 2004).

```
FROM Project//ProjectClauses as a
WHERE a@old subset a@new
MATCH: InteractionName;
```

The first statement is a simple XPath-like statement, which defines a variable named `a` and binds it to all nodes in the AST trees whose node type is `ProjectClauses` and is the child of a `Project` node. Since we are comparing a pair of query (q_1, q_2), `a@old` and `a@new` references the project clauses of q_1 and q_2 respectively. The second statement is a boolean expression, and the third statement matches the pair of queries to the interactions stated if the second statement returns true.

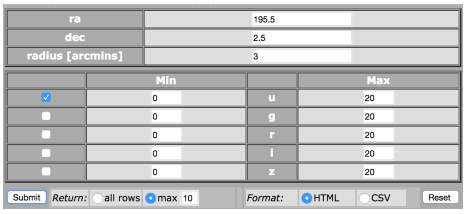
This program returns sets of query pairs, such that each query in the pair can be transformed into the other via the specified transformation. To infer an interface from those results, we operate in two steps. First we build an *transformation graph*, in which each vertex represents a query and each edge a transformation. Then, we assign widgets to the transformations. For instance, we may use text boxes, drop down lists or radio buttons to express changes in a `Project` statements. The challenge is to cover as many queries as possible, while keeping the interface simple.

Technically, we express widget assignment as a knapsack problem. We assume that each widget w has a predefined visual complexity $C_{vis}(w)$ and activation cost $C_{act}(w)$. Furthermore, we suppose that each widget can cover a known range of interactions (i.e., edge labels in the interaction graph). We seek the set of widgets that allows the users to jump from any query of the log to any other with the smallest possible effort. If $Q_i \in \mathcal{L}$ represents the programs in the log, $w \in \mathcal{G}$ the widgets on the interface, $penalty$ and max_{vis} two parameters, we express the problem as follows:

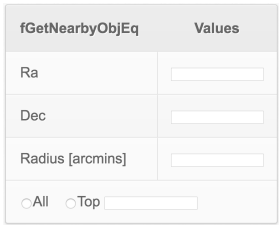
$$\text{argmin}_{\mathcal{G}} \quad \frac{1}{|\mathcal{L}^2|} \cdot \sum_{Q_i, Q_j \in \mathcal{L}^2} \min_{w \in \mathcal{G}} \begin{cases} C_{act}(w) \text{ if } w \text{ covers } (Q_i, Q_j) \\ penalty \text{ otherwise} \end{cases}$$

$$s.t. \qquad \sum_{w \in \mathcal{G}} C_{vis}(w) < max_{vis}$$

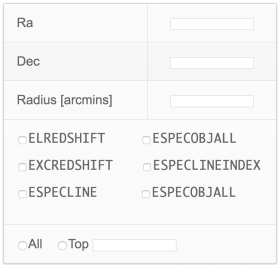
The objective function describes the average user cost necessary to navigate between two log entries. If several widgets cover the same pair of programs (Q_i, Q_j), then the function picks the best one. If no widget can express the transformation, then the function applies a penalty. The constraint enforces that total complexity of the interface stays lower than max_{vis}. We currently solve the problem



(a) Original SDSS interface.



(b) Generated interface - prefers simplicity.



(c) Generated interface - prefers coverage.

Figure 7: Original and generated interfaces, based on the SkyServer query log.

with a greedy heuristic. By modifying the parameters $penalty$ and max_{vis}, as well as the relative costs of the widgets, we can produce a wide range of candidate interfaces and chose the one that fits the use case best.

Figure 6 depicts a sample of the transformation graph of SDSS, that we obtained with 8 hand coded transformation queries. Each edge color denotes an interaction type. Typically, we find that this graph is extremely dense: the two most frequent interactions cover 12% and 70% of our sample query log, respectively. Figure 7 shows the original SDSS interface, as well as two mock-ups that we generated from the output of our system. It turns out that only a subset of the current interface is used for any given analysis session, and thus it can be drastically simplified.

Our plan is to explore richer graph structures, penalty functions, and interactions across a number of languages. In addition, there is opportunity to combine Precision Interfaces with Scalable Interaction analysis to generate analysis-specific, performance-aware interfaces in real-time.

4. CONCLUSION

Database technology has long benefitted from applying declarativity and data independence ideas to novel domains—from stream processing, to web content, to sensor networks. The ideas in DVMS aim to bring user experience, interface design, and perception into the declarative mindset. The introduction presented existing challenges faced by interactive visualization developers that can benefit from such an approach. Likewise, focusing on interactive visualizations bring forth novel constraints and solutions that can help push the database community forward.

5. ACKNOWLEDGEMENTS

This research was supported by NSF IIS #1564049

References

[1] Rio 2016 - interactive stories. In *The New York Times*, 2016.

[2] Rstudio ggviz. http://ggvis.rstudio.com/, 2016.

[3] Tensorboard: Visualizing learning. In *TensorFlow*, 2016.

[4] D. Alabi and E. Wu. PFunk-H: Approximate query processing using perceptual models. In *HILDA*, 2016.

[5] F. D. Albareti et al. The thirteenth data release of the sloan digital sky survey: First spectroscopic data from the sdss-iv survey mapping nearby galaxies at apache point observatory. *ArXiv e-prints*, 2016.

[6] T. Althoff, X. L. Dong, K. Murphy, S. Alai, V. Dang, and W. Zhang. Timemachine: Timeline generation for knowledge-base entities. In *KDD*, 2015.

[7] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, 2003.

[8] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *SIGMOD*, 2016.

[9] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Vis*, 2005.

[10] R. A. Becker and W. S. Cleveland. Brushing scatterplots. In *Technometrics*, 1987.

[11] C. M. Brown. *Human-computer interface design guidelines*. Intellect Books, 1998.

[12] S. Carter, D. Ha, I. Johnson, and C. Olah. Experiments in handwriting with a neural network. http://distill.pub/2016/handwriting/, 2016.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD*, 2003.

[14] R. Chang, M. Ghoniem, R. Kosara, W. Ribarsky, J. Yang, E. Suma, C. Ziemkiewicz, D. Kern, and A. Sudjianto. WireVis: Visualization of categorical, time-varying data from financial transactions. In *VAST*, 2007.

[15] Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/, 2016.

[16] J. Clark and S. DeRose. XML path language (XPath) version 1.0. *W3C recommendation*, 1999.

[17] Crossfilter. http://square.github.io/crossfilter/, 2015.

[18] R. Ebenstein, N. Kamat, and A. Nandi. FluxQuery: An execution framework for highly interactive query workloads. In *SIGMOD*, 2016.

[19] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. Vistrees: fast indexes for interactive data exploration. In *HILDA*, 2016.

[20] W. Eugene, B. Leilani, and R. M. Samuel. The case for data visualization management systems. In *VLDB*, 2014.

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

[22] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. *TKDE*, 2015.

[23] J. Harper and M. Agrawala. Deconstructing and restyling d3 visualizations. In *UIST*, 2014.

[24] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *SoCC*, 2012.

[25] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.

[26] M. Herschel and M. Hlawatsch. Provenance: On and behind the screens. In *SIGMOD*, 2016.

[27] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual debugging techniques for reactive data visualization. In *EuroVis*, 2016.

[28] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. In *VLDB*, 2008.

[29] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *ICDE*, 2014.

[30] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: multitouch gestures as regular expressions. In *CHI*, 2012.

[31] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. In *EuroVis*, 2013.

[32] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. In *Vis*, 2014.

[33] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, 2013.

[34] P. Longhurst, K. Debattista, and A. Chalmers. A GPU based saliency map for high-fidelity selective rendering. In *AFRIGRAPH*, 2006.

[35] J. H. Murray. *Inventing the medium: principles of interaction design as a cultural practice*. Mit Press, 2011.

[36] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. In *VLDB*, 2014.

[37] C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI*, 2000.

[38] Oracle. Oracle endeca information discovery: A technical overview. Technical report, 2014.

[39] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *TVCG*, 2017.

[40] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. In *SIGMOD*, 2015.

[41] E. D. Ragan, A. Endert, J. Sanyal, and J. Chen. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. *TVCG*, 2016.

[42] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[43] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *CACM*, 2009.

[44] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. In *VLDB*, 2015.

[45] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *EuroVis*, 2014.

[46] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *TVCG*, 2017.

[47] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. In *InfoVis*, 2015.

[48] SDSS Query Logs. http://cluster.ischool.drexel.edu/~jz85/SDSSLogViewer/data.html, 2004.

[49] B. Shneiderman. Response time and display rate in human performance with computers. *CSUR*, 1984.

[50] Splunk. http://www.splunk.com.

[51] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *TVCG*, 2002.

[52] D. F. Swayne, D. T. Lang, A. Buja, and D. Cook. GGobi: evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 2003.

[53] Tableau. http://www.tableausoftware.com.

[54] T. Tylenda, M. Sozio, and G. Weikum. Einstein: Physicist or vegetarian? summarizing semantic type graphs for knowledge discovery. In *WWW*, 2011.

[55] B. Walenz, J. Gao, E. Sonmez, Y. Tian, Y. Wen, C. Xu, B. Adair, and J. Yang. Fact checking congressional voting claims. In *Computation+Journalism Symposium*, 2016.

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

[57] A. Wilhelm. User interaction at various levels of data displays. *CSDA*, 2003.

[58] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[59] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *PVLDB*, 2013.

[60] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, 2013.

[61] Y. Wu, J. M. Hellerstein, and E. Wu. A devil-ish approach to inconsistency in interactive visualizations. *HILDA*, 2016.

[62] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using GUI screenshots for search and automation. In *UIST*, 2009.

[63] J. S. Yi, Y. a. Kang, J. Stasko, and J. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *TVCG*, 2007.