(Demo) Physical Visualization Design

Lana Ramjit
lana@cs.ucla.edu
UCLA

Zhaoning Kong
jonnykong@cs.ucla.edu
UCLA

Ravi Netravali
ravi@cs.ucla.edu
UCLA

Eugene Wu
ewu@cs.columbia.edu
Columbia University

ABSTRACT

We demonstrate PVD, a system that visualization designers can use to co-design the interface *and* system architecture of scalable and expressive visualization.

1 INTRODUCTION

Building interactive data visualizations is hard. It requires expertise spanning human-computer interaction, networking, and database optimization. Visualization designers need to ensure that the interface's visual layout is expressive enough to accomplish the desired user tasks. At the same time, designers also need to make architectural and systems optimization decisions in order to ensure that the interface is responsive in the face of large and growing data sizes.

The processes of designing an effective interface and developing a responsive architecture are intertwined: the interface and interaction design determine the data flows expressible by the user, while the architecture design determines the scale at which these data flows can execute quickly enough. For instance, an interface consisting of a single small dropdown menu can ensure interactive speeds by pre-computing and caching the query results associated with each of the options. However, this strategy fails when adding an interaction such as free-text search, which would require an inordinate amount of pre-computation storage, and thus necessitates a different architectural design strategy.

The complexity of such design decisions poses a major practical challenge because creating new visualization interfaces is not a one-shot process. Instead, designers iteratively create prototypes, using the feedback from their intended users to refine the design, add new views and interactions, and ensure that the interactions are sufficiently responsive. If the data size is negligible, then the designer can focus solely on interface design, which is well-supported by existing visualization [9] and design [2, 6] tools. However, if the datasets are large, then simply creating a prototype requires setting up a server that connects to a data management system, and making physical database design and caching optimizations so the prototype is responsive. Even if the designer is capable of this engineering work, the tremendous engineering cost can "lock-in" the designer to early architecture decisions.

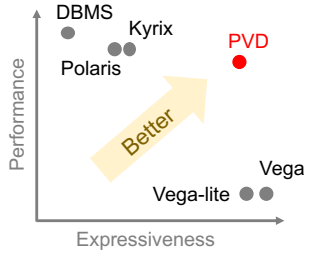

Figure 1: Current visualization frameworks trade-off expressiveness and performance.

There is a need for tools that support the rapid co-design of the visualization interface and the system architecture. Unfortunately, existing tools primarily focus on one of the two aspects (Figure 1). At the extremes, visualization libraries like Vega-lite [9] help accelerate client-side visualization design, whereas database tuners [1, 3] optimize the physical data layouts but are agnostic to the application interface design. Scalable visualization frameworks like Kyrix [10], Falcon [8], and Polaris [7] make specific architectural decisions that limit the designer to a subset of designs or interactions that the architecture can efficiently express, e.g., pan/zoom or brushed linking.

To overcome these limitations, PVD is a co-design tool that helps visualization designers rapidly iterate through the combinatorial space of interface designs and physical layouts. The key challenge is in identifying the appropriate abstraction for specifying PVD's inputs. It must be low-level enough to express a wide range of visualization interfaces, yet high-level enough to enable effective optimization. To address this tension, PVD models the interface as a set of data flows (SQL queries) that are structurally transformed and executed in response to user interactions.

Thus, PVD takes as input a specification of the interface design, architectural optimization techniques, constraints on the available resources, and expectations of interactivity

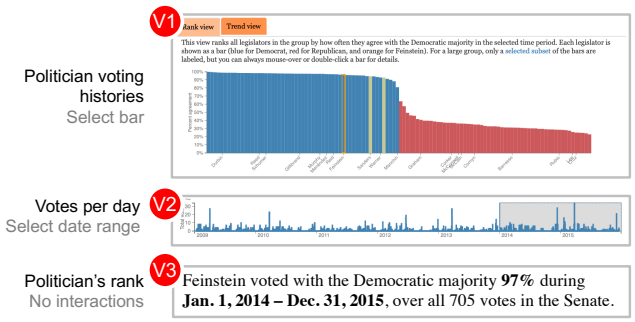

Figure 2: Three example iCheckuClaim charts (views).

and responsiveness. For instance, the designer may specify that the dataset can vary from 2MB to 2GB, server and client memory are 2GB and 150MB respectively, and that she expects interactions to be serviced within 50ms. PVD then outputs the expected response times of the interactions in the interface and, if the constraints cannot be met, recommends ways to modify the interface or architectural designs. The estimated response times are directly simulated in the design interface, so the designer can directly experience the effects of different architecture choices across the system.

In this demonstration, users will use PVD to interactively build a scalable visualization interface by adding visual components and interactions, and see how even small interface design choices affect the utility of different architectural designs. At each step of the design process, users will receive immediate feedback on the visualization's performance and design from PVD. Once a user is satisfied, PVD will instantiate and deploy the designed interface.

2 USE CASE

iCheckuClaim [11] is a web-based visualization application developed by database researchers at Duke University. Users explore and contextualize U.S. politician voting records as compared to peer groups (e.g., Republican/Democratic majority, the President, etc). We describe the interface design and architecture of a subset of 3 visualizations (called "views") in the main interface (Figure 2).

2.1 Interface Design

(V1) Ranked Politicians Histogram: The sorted bar chart lists all politicians within a demographic group (e.g., all senators, female representatives) along the x-axis. The y-axis shows the percentage of votes cast by a politician that aligned with the position of a user-selected peer group. The user picks each of the two groups from a pre-defined list. The orange bar is the currently selected politician (e.g., Senator Feinstein).

(V2) Vote Count Histogram: This bar chart shows the total number of votes per week over five years. Users can select a date range (the shaded region), which will update the percentages in V1 based on votes in the selected range, and also update V3 described next.

(V3) Politician Text Summary: This component describes a specific politician's rank and percentage of agreement within the currently selected time window. This is updated when a new bar (representing a new politician) is selected from V1 and when the selection in V2 changes.

2.2 System Architecture

iCheckuClaim uses a client-server architecture. The backend data store is a ReDis instance which stores the raw voting data. Additionally, prefix-sum indexes for certain common peer groups (such as the President and political party majorities) are pre-computed and cached in-memory. Other peer groups compute a prefix-sum index on the fly. Requested indexes are sent to the client.

The Javascript client caches all data received from the server, and reloads the page when a new peer group is selected. As users interact with the interface, event-handlers interpret user interactions and decide whether to update the interface using the client cache or send a server request. The cached prefix-sum index can recalculate a politician's voting behavior in constant time as the user selects new date ranges or different politicians.

2.3 Challenges

Even in this simple interface, subtle interface design decisions have considerable affect on the architecture. For instance, pre-defining the peer groups limits the user's choices but enables pre-computation. Rendering V1, V2, and V3 on the same page implies that the user will expect V1 to quickly update as the user creates and resizes a selection in V2. Further, the designer must choose whether the selection should continuously update V1 and V3 as it is manipulated, or only when the user finishes the selection interaction.

Each of these choices adds or removes architectural requirements in terms of the data structures, caching, and data placement choices that must be made to meet the user expectations. However, choosing to materialize an index to accelerate the date range interaction can reduce the resources available for pre-computing and optimizing interactions in V1. Designers may ultimately need to choose which interactions to prioritize in response to limited resources.

The combination of interfaces and optimizations is too large to manually search. We next outline how PVD models this as a constraint-based optimization problem to help designers make informed trade-offs.

3 PVD OVERVIEW

PVD follows the principles of data independence: it combines a declarative specification of the core data-flows and interactions used in the interface, with an optimizer that solves an architectural design problem within resource and interactivity constraints. We describe each below, and the user-facing design interface in the next section.

3.1 Visualization Specification

PVD models each view as a SQL query, and each interaction as a directed edge from a source view (that the user interacts with) to a target view (that updates in response). We based this model on prior work in interface generation [12].

A view consists of a SQL query and a visualization rendering specification. For example, V2 in Figure 2 computes a given chamber's votes per day as an aggregation query:

```
SELECT date as d, COUNT(vote_id) as vote_cnt
FROM votes v
WHERE v.vote_chamber === chamber
GROUP BY v.date
```

The following visual encoding spec maps query attributes to bar chart properties: `mark=bar`, `d→xaxis`, `vote_cnt→yaxis`.

An interaction is composed of user interaction data, and how it modifies the structure of the target view's query. Each user interaction (e.g., selecting an option in a dropdown, dragging a selection box, etc) exposes a record that contains the event and data information (e.g., the option that was selected, the selection box's bounds as dates). Query modifications may be simple—such as changing a query parameter—or change the entire query structure. The main requirement is that the transformed query remains schema compatible.

For example, the following specifies a 1D brush in V2, and that it should update V1's query by setting its date range filter based on the brush's range. More complex structure transformations are expressed as abstract syntax tree transformations that are extracted from query examples [5, 12].

```
interaction:              update v1:
  view: v2                  SET v1.date1 TO 1dbrush.minx
  type: 1dbrush             SET v1.date2 TO 1dbrush.maxx
```

This graph representation encapsulates the data-flows needed to drive the visualization, yet gives the designer flexibility in terms of the *visual design* of the views, layout, and interactions. For instance, they may use Vega-lite, or any other visualization library, to render query outputs—we assume that this rendering process is not the dominant overhead in interaction response times. At the same time, this graph compactly represents all possible queries expressible by the interface. We now describe our current method for using this graph for architectural optimization.

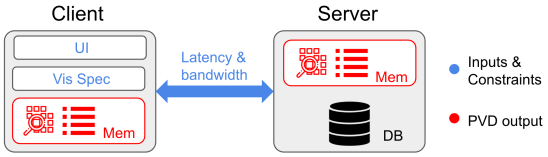

Figure 3: A template of the client-server architecture that PVD outputs. Blue components are inputs controlled by the developer, while red components are filled in by PVD.

3.2 Optimization and Latency Estimate

PVD takes as input the visualization specification, resource constraints, and network characteristics. Assuming the client-server template architecture shown in Figure 3, PVD takes a sample interaction trace from the interface and generates a query workload. It then recommends what data structures to create and where to place them. Based on the recommended architecture and available cost models PVD also estimates the latency for each visual component, binning them into immediate (10-100ms), fast (100-500ms),and slow (500+ ms).

PVD provides an extensible library of visualization optimizations and data structures. While PVD currently supports tree, hash, and prefix-sum indexes, data cubes, and pre-computation, developers can add custom optimizations by providing two functions. $\text{check}_k()$ ensures that a query is valid for a given optimization. It does so by checking that an interaction's query transformation specification is applicable to the optimization. For instance, the 1dbrush interaction modifies a range predicate in V1's query and thus acceptable for tree indexes, and V1's count aggregation makes it acceptable for prefix-sum indexes. It also needs to extract a query template signature so that queries with different structure will map to different e.g., prefix sum indexes.

$$\text{check}_k(\text{i}) \rightarrow (\text{T/F}, \text{signature})$$

The second function, estimate_k takes as input a query q and database statistics (e.g., cardinalities, attribute distributions), and uses optimization-specific cost models to estimate the size of the data structure and latency if the optimization is applied to the q:

$$\text{estimate}_k(\text{q}, \text{stats}) \rightarrow (\text{size}, \text{latency})$$

Some optimizations require designer input when it returns results that are not strictly equivalent to the query result. For example, sampling introduces uncertainty in the results. In this case, the optimization needs to quantify the way that the results may diverse from the true results, so the developer has the option to specify the degree of e.g., uncertainty they are willing to accept as an additional constraint.

Finally, PVD must select a set of optimizations and a data placement policy that best reflects the developer constraints. Our current approach is inspired by existing physical database design solutions [4]. We enumerate each combination