Optimizing Dataflow Systems for Scalable Interactive Visualization

JUNRAN YANG, University of Washington, USA
HYEKANG KEVIN JOO, Carnegie Mellon University, USA
SAI YERRAMREDDY, University of Maryland, USA
DOMINIK MORITZ, Carnegie Mellon University, USA
LEILANI BATTLE, University of Washington, USA

Supporting the interactive exploration of large datasets is a popular and challenging use case for data management systems. Traditionally, the interface and the back-end system are built and optimized separately, and interface design and system optimization require different skill sets that are difficult for one person to master. To enable analysts to focus on visualization design, we contribute VegaPlus, a system that automatically optimizes interactive dashboards to support large datasets. To achieve this, VegaPlus leverages two core ideas. First, we introduce an optimizer that can reason about execution plans in Vega, a back-end DBMS, or a mix of both environments. The optimizer also considers how user interactions may alter execution plan performance, and can partially or fully rewrite the plans when needed. Through a series of benchmark experiments on seven different dashboard designs, our results show that VegaPlus provides superior performance and versatility compared to standard dashboard optimization techniques.

CCS Concepts: • **Information systems** → **Data management systems**; • **Human-centered computing** → **Visualization systems and tools**; **Interactive systems and tools**.

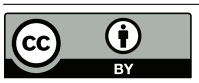
Additional Key Words and Phrases: data analytics, scalable visualization

ACM Reference Format:
Junran Yang, Hyekang Kevin Joo, Sai Yerramreddy, Dominik Moritz, and Leilani Battle. 2024. Optimizing Dataflow Systems for Scalable Interactive Visualization. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 21 (February 2024), 25 pages. https://doi.org/10.1145/3639276

1 INTRODUCTION

Interactive data visualization is essential for understanding, manipulating and presenting complex datasets [9, 60]. Specification languages such as D3 [7], Plotly [23] and Vega [49] make the process of designing interactive visualizations more systematic, precise and accessible to both novices and analysts. Meanwhile, they are also the building blocks of all visualization systems. For example, Forcache [1] and Kyrix [57] are implemented using D3 [7] while Falcon [37] and Voyager [63] are built on top of Vega/Vega-Lite [48, 49]. Hence, enhancing visualization languages can not only extend their usability but also improve all of the systems they support, leading to a cumulative impact on the data visualization ecosystem. However, these languages are not designed to optimize data queries, and, as a consequence, they fail to match the data processing performance of even the most basic database systems [36].

Authors' addresses: Junran Yang, junran@cs.washington.edu, University of Washington, Seattle, USA; Hyekang Kevin Joo, Carnegie Mellon University, Pittsburgh, USA, kevinjoo@andrew.cmu.edu; Sai Yerramreddy, University of Maryland, Collkge Park, USA, saiyr@umd.edu; Dominik Moritz, Carnegie Mellon University, Pittsburgh, USA, domoritz@cmu.edu; Leilani Battle, leibatt@cs.washington.edu, University of Washington, Seattle, USA.



ACM 2836-6573/2024/2-ART21
https://doi.org/10.1145/3639276

A natural next question is: why not simply add classic data management optimizations and/or cost models to visualization languages? Prior works show how naive applications of classic database optimization methods fail to support interactive visualization scenarios [1, 2, 19, 30, 44]. In contrast, visualization- and interaction-aware optimization approaches are more effective for scaling up visualization systems [5]. Moreover, with the huge diversity in possible visualization and dashboard designs, system configurations, and user interaction behaviors, it is difficult to hand-tune classic cost models for selecting efficient optimization plans in each analysis scenario. While many query optimization techniques have also been proposed to support specific interface or interaction designs [5], they often fail to translate to new visualization interfaces. This makes developing an optimizer a major challenge for big data visualization and exploration systems. Furthermore, given that our target users are not always system builders or DBMS administrators, *an ideal solution is to build dynamic optimizations directly into visualization languages*, so that the users can avoid bespoke optimization techniques.

In this paper, we explore the design space of possible optimization strategies for Vega [49], a popular visualization language. To do this, we first extend Vega to coordinate execution with a back-end DBMS such as PostgreSQL [43] or DuckDB [45]. Then, we explore the space of possible optimization strategies by training machine learning models to predict efficient Vega+DBMS execution plans among hundreds of candidates derived from a variety of dashboard configurations. By inspecting what these models learn, we contribute robust heuristics towards the development of effective cost models for Vega. Based on our findings, we introduce VegaPlus, a system that automatically optimizes dashboards implemented in Vega by (a) rewriting Vega's data transformations as corresponding SQL queries and (b) selecting efficient plans for executing these queries across the client-side Vega runtime and server-side DBMS.

A declarative Vega specification often contains a data transformation pipeline, i.e., a visualization query that generates a *dataflow graph* as the execution plan. We contribute a **new query rewriting approach for Vega** that augments selected dataflow operators to emit a SQL query that performs equivalent data manipulation operations (e.g., `filtering`, `binning`, and `aggregation`). In this way, we can augment Vega to leverage the computational strengths of relational DBMSs like PostgreSQL [43] and DuckDB [45] with minimal changes to the user's visualization workflow.

However, naively pushing all computation to the DBMS can inadvertently *increase* system latency, since it completely ignores available computing resources on the client and may introduce unnecessary round trips across the network. Furthermore, the DBMS is unaware of how user interactions may alter query execution across Vega dataflows, hindering its ability to optimize these dataflows on its own. For example, interactive filters (e.g., sliders, brushes) may or may not generate filter predicates on emitted queries, depending on the user's interaction choices. In response, we explore a range of techniques for selecting efficient query execution plans in an interaction-aware way. Specifically, we contribute **a collection of machine learning models to predict efficient plans** given metadata about dataflow structure, dataset characteristics, and anticipated execution costs for static *and* interactive visualization scenarios. Then, we analyze these models to contribute **general-purpose heuristics** that can be used to develop robust cost models for Vega, which we refer to as *heuristic-based models*. Furthermore, we extend these optimizations to exploit repetition in user interaction behaviors by checking when query plans may benefit from previously cached results, inspired by prior work [1, 59].

To identify the best models and heuristics for selecting efficient execution plans, we contribute **a Vega benchmark suite** that simulates interactions with seven different Vega dashboard templates. We use this benchmark to test the efficacy of our plan selection models across different visualization designs, input datasets, and interactive scenarios, inspired by prior work on visualization benchmarking [3, 15].

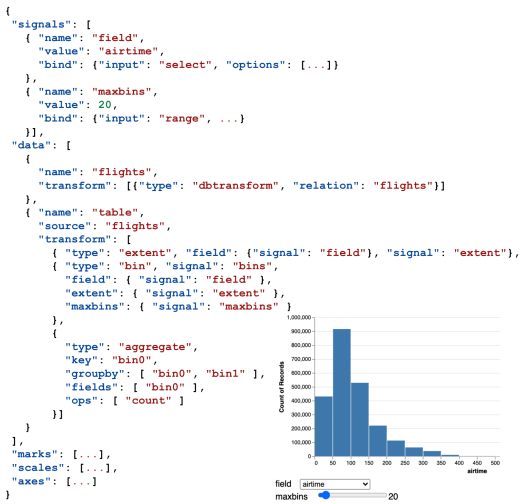


Fig. 1. A histogram example with dynamic queries. Users can select a data field from the drop-down menu, or adjust the bin size with the slider.

Our techniques can also generalize beyond Vega to any visualization language that compiles into dataflow graphs. Given a dataflow graph from another language, our query rewriting methods can easily be extended to map detected data transformation nodes to corresponding SQL queries, which can then be passed directly to the VegaPlus optimizer. These techniques can be particularly useful in development environments that contain significant information about the client but limited server-side control, e.g., in online notebook environments such as Observable [55].

In this paper, we make the following contributions:

- Automated query rewriting for the Vega visualization language, which translates data transformations from Vega specifications into SQL queries to be executed on a connected DBMS.
- A systematic analysis of Vega's query optimization space, using machine learning models to navigate hundreds of viable execution plans that balance data processing between the client dataflow in Vega and an external DBMS.
- Comparison of multiple optimizer designs for supporting interactive visualization of large datasets, based on the results of our aforementioned analysis.
- A new Vega performance benchmark and evaluations demonstrating how the proposed optimizers scale up and speed up visualization processing for a wide range of existing designs compared to Vega.

2 BACKGROUND

VegaPlus aims to scale up interactive visualizations generated with Vega for large data. Here, we provide a brief overview of Vega and its dataflow system.

Vega Dataflow. The Vega runtime parses specifications in JSON to generate interactive, web-based graphics. To efficiently process and route data to relevant visualization components and layers, Vega adopts a dataflow compilation and execution model [49]. Dataflow systems are of particular interest when performing interactive, incremental stream processing in a distributed environment [11, 20, 38, 39]. It is also a common data model in visualization systems (e.g., Vega [49], VTK [50]) where its operators form a directed graph.

As a declarative language, Vega decouples specification from execution, allowing customization and optimization for data processing in its dataflow system. Given a user's declarative specification, Vega automatically constructs the corresponding dataflow graph. When executed, the dataflow graph computes a series of data transformations through known operators (e.g., filter, map, aggregate), where data records are processed by each operator as they pass through the graph. Finally, the computed data is mapped to visual encodings and positions. As an example, the Vega specification in Figure 1 uses the `bin` and `aggregate` transforms to calculate the bins and counts for the rendered histogram. Note that the instantiated dataflow graph at runtime may include additional internal operations that are not declared in the user's specification to support data processing, such as copying or sorting the data to facilitate the creation of axes, scales, etc.

In VegaPlus, we propose new features to enable relational DBMSs to participate in the computation of Vega operators while preserving the original Vega dataflow structure. In this way, we enable users to connect visualization languages with data processing systems that they may not otherwise benefit from.

Data Pipeline. The `data` component (i.e., data pipeline) in a specification is an array of `data entries`, and each data entry may contain a series of Vega `transforms` in the transform array, as in Figure 1. Each data entry points either to another data entry's output or the raw data as source. In the instantiated dataflow, corresponding operations maintain the same relative order as in the specification, even though internal operators are added. The data pipeline is transformed into the dataflow, which is a directed acyclic graph (DAG). Figure 3 demonstrates a simplified version of the dataflow: the root nodes are signals and the data source references the DBMS tables, while non-root nodes are the transform operators.

From Specification to Dataflow Graph. Vega `Transforms` are the operators inside the dataflow graph for data processing and transformation. Each transform operator takes data tuples as input, performs certain computational operations on them, and then generates new data tuples as output. Vega includes a variety of common transform types that are particularly useful in constructing visualizations [49]. Filtering, binning and aggregation are common examples of transform operations. Vega Transforms are typically specified within the transform array of a data definition. At compilation time, Vega instantiates a dataflow where the transform operators form a path in the order they are specified. Additional internal operators are added in between to calculate needed statistics or propagate updates. However, since the operators are evaluated in topological order, the way the data is processed is entirely decided by the specifications set by the user. Although SQL queries share the same declarative nature as Vega's visualization specifications, a DBMS typically rearranges and optimizes how the data are processed, producing a more efficient query execution structure overall. Hence, we propose techniques to partition the dataflow operations for execution within a DBMS while maintaining the relative orders. However, we have also designed our dataflow rewriting methods to support extensions in the future.

Vega Parameters & Signals. Parameters that specify an operator can either be fixed values or live references to other operators. Interaction events can update operator parameters or data inputs, and the changes are only re-evaluated by the operators downstream to the update. Further, users can track the interaction state in the Vega dataflow by specifying `signals`, i.e., special variables in the Vega specification used to capture the output of triggered interactions. To this end, signals can be used to dynamically parameterize visual encoding properties or transform parameters via expression languages. For example, in Figure 1, two signals are specified: the first signal binds selections from the `field` dropdown menu to the `extent` transform in the specification and the second signal binds values from the `maxbins` slider to the number of bins calculated in the `bin` transform. Together,

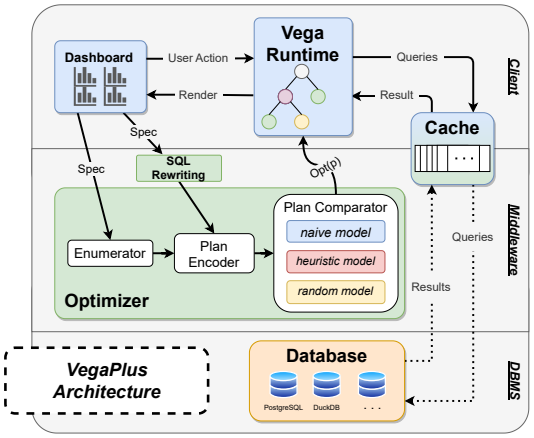


Fig. 2. The architecture diagram.

these signals control how the x-axis and bins (i.e., rectangles) are rendered in the histogram. By extension, Vega signals become a form of data provenance that can be monitored by VegaPlus to detect changes in execution flow, e.g., to detect changes in the grouping/binning predicates in Figure 1 when the user drags the slider, which would trigger recalculating the histogram bins.

Example 2.1. Putting everything together, the example in Figure 1 shows a histogram visualization alongside its Vega specification. Its source dataset consists of flight arrival and departure details for all commercial flights in the USA from 1987 to 2008 [40]. To explore the data distribution in terms of each data field, users can select the target data field from a drop-down menu and use the slider to find a desired binning granularity to summarize the record count. Although histograms are a common visualization type in data analysis, multiple operations are carried through when users interact with the chart. An `extent` *transform is performed to calculate the x scale extent (minimum and maximum), which is also required by the* `bin` *transforms to calculate the binning buckets' start and end values. Other than the extent, the binning transform also needs the signals from both the* `maxbins` *slider and* `field` *drop-down menu. After the source data is distributed into discrete buckets, an* `aggregate` *transform counts the items inside each bucket. We use this example in the following sections to explain our optimization flow.*

3 SYSTEM OVERVIEW

VegaPlus is comprised of three layers (see Figure 2): a client-side layer for rendering the visualizations, a server-side middleware layer for optimization, and a remote DBMS for scalable data processing. In this section, we summarize the functionality of each layer.

Client-Side Vega Runtime. The Vega runtime generates the original Vega dataflow graph that renders the target visualization(s). The Vega runtime also manages any signals (i.e., interaction triggers) defined in the Vega specification, which VegaPlus needs to assess how interactions may alter the execution of the corresponding Vega dataflow graph. We have also augmented the Vega runtime to use specialized dataflow nodes that enable VegaPlus's SQL query rewriting functionality. Please see Section 4 for more details.

Server-Side Optimizer. The optimizer is responsible for evaluating a given Vega dataflow graph, enumerating possible execution plans involving client and/or server resources, and selecting an efficient plan. To enumerate plans, the optimizer must coordinate with the client-side Vega runtime

and (possibly remote) DBMS. To evaluate plans, the optimizer translates each plan into a vector encoding that can be recognized by machine learning models. Then, models are used to predict the most efficient plan. Finally, these predictions are used to dispatch queries either to the Vega runtime for local execution or to the DBMS for server-side or remote execution. We describe the optimizer further in Section 5.

DBMS.. VegaPlus gives users the option to share connection information for a DBMS within a Vega specification. This DBMS can be local, e.g., running on the client or even running in the browser, or it can be remote. VegaPlus uses this DBMS as a back-end for data processing. Specifically, any transforms from the Vega dataflow that are selected for SQL-based execution by the optimizer will be processed using the DBMS. To do this, VegaPlus issues the target queries to the DBMS using the specified connection, retrieves the results, then passes them back to the Vega runtime to be rendered. VegaPlus also uses the DBMS's plan analyzers to optimize Vega dataflow queries. For example, VegaPlus leverages the DBMS explain command to estimate execution costs. Please see Section 5 for more details.

4 QUERY REWRITING: FROM VEGA OPERATORS TO SQL QUERIES

To facilitate interactive exploration in visualizations, transform operations are often one of the most time- and resource-intensive components. To address this problem, the query rewriter builds a bridge for VegaPlus to offload computationally-intensive Vega transforms as SQL queries executed on the backend DBMS. Specifically, we parse the Vega JSON specification and traverse the data pipeline to rewrite Vega transform operations into SQL queries. Then, we replace those rewritten operators with custom `VegaDBMSTransform` (VDT) operators. At runtime, when a VDT node is triggered, it builds and issues the corresponding SQL query and fetches the results via the middleware server.

Candidate Transforms for Rewriting. The VegaPlus query rewriter identifies applicable Vega transform nodes in the dataflow graph, such as extent (i.e., range values), bin, aggregate, filter, collect (i.e., sorting), projection and stack (i.e., window functions), automatically parses the transform parameters (if they exist), and translates the operation to a SQL query string builder using pre-defined templates implemented for each transform type. We targeted Vega transforms that match the select, project, group, and aggregate operations typically observed in OLAP queries to support offloading these operations to applicable DBMSs. That being said, these SQL query builders often contain dynamic variables that are placeholders for signals or the output of parent nodes earlier in the dataflow. Thus, VegaPlus must balance code and SQL query integration in a way that other visualization or database contexts may not. For example, VegaPlus tracks how each node's dependencies are evaluated and propagated through the dataflow graph, allowing it to update the corresponding transform node with a complete SQL query when the holes in the query builder are filled. Translation of some common transform operators are straightforward due to the structured JSON format and clearly defined parameters such as the "groupby", "ops" and "fields" for `aggregate`. The `filter` transform is more complex to translate. It contains a predicate expression in a JavaScript-like expression language (e.g., `{"type": "filter", "expr": "datum.delay > 10 && datum.delay < 30"}`), which evaluates to true or false to filter each data object. We parse the expression string into an Abstract Syntax Tree (AST) to generate a WHERE clause in SQL (e.g., WHERE delay > 10 AND delay < 30). Note that there is not always a one-to-one mapping between these Vega expressions and SQL. When an equivalent SQL predicate is not found, we fall back to native execution in Vega.

Efficient Transfers Between Vega and a DBMS. To automate two-way communication between the client and DBMS, VegaPlus replaces the pipelines of Vega transforms with `VegaDBMSTransform`

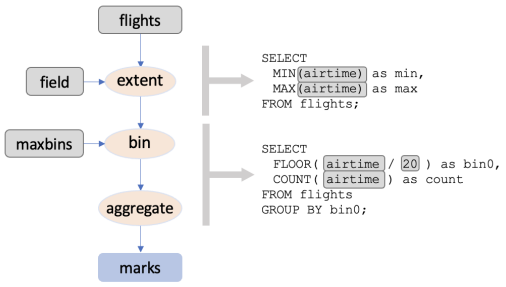


Fig. 3. One possible way to rewrite the queries for the histogram example in Figure 1. The queries are parameterized by the input signals. The augmented nodes (VDTs) will re-evaluate and send queries to the DBMS when signals update.

(VDT), a custom transform type that is responsible for building and executing SQL queries against a DBMS. VDTs act as data sources in the data entry of a specification as in Figure 1.

VDTs are atypical Vega transforms in that they do not take any input data tuples from the upstream Vega dataflow; rather, the input data tuples can be thought of as the tuples from the DBMS (e.g., PostgreSQL) in relation to the VDT targets. When a VDT's transform function is executed as part of dataflow graph execution in the browser, it sends an HTTP request containing an SQL query to a middleware server and waits for the results. The middleware server then forwards the query to a DBMS (PostgreSQL in this case) and then returns the query results to the browser. When the VDT in the browser receives these results, it emits them for propagation to downstream nodes in the Vega dataflow graph. It is important to note that VDT transform functions are blocking, synchronous operations. So far, we have assumed query rewriting based on individual transform operators. However, each VDT requires the query result to be returned to the client and stored in the dataflow graph, which leads to extra data transfers and memory consumption. To address this challenge, we support recursive rewriting of multiple transforms into a single nested query, i.e., batching transforms to reduce round trips. For example, an upstream transform can be rewritten as a subquery in downstream translation. Further, we also support rule-based query rewriting to transform nested batch queries into a more readable format, as shown in Example 4.1. By default, our HTTP connector supports JSON, which requires client-side decoding and leads to large serialization overhead. To further reduce network transfer costs, VegaPlus encodes query results using the binary Apache Arrow format [47].

Example 4.1. In Figure 3 we illustrate one way to rewrite the example specification in Figure 1. Suppose the VegaPlus optimizer emits an execution plan that rewrites the original `extent`, `bin`, *and* `aggregate` *transforms using two VDTs: one for the* `extent` *and the other for* `bin` *and* `aggregate`. *The* `extent` *query is in a separate VDT because its output is referenced by the* `scale` *and the* `bin` *operator as a* `signal`. *At dataflow evaluation time, the* `extent` *VDT's evaluation function sends its SQL query to the middleware layer for execution. Once executed by the DBMS, the middleware sends the query results back to the Vega runtime, which get propagated through the dataflow graph to the* `bin-and-aggregate` *VDT, where the bin's step size is calculated to complete the query string. Then, this query is evaluated in the same manner as the previous* `extent` *VDT.*

5 OPTIMIZATION

VegaPlus optimizes visualization dataflow execution by finding the most suitable plan to execute the corresponding Vega transforms and propagating the results to the Vega renderer. To achieve

this, the optimizer enumerates all valid possibilities for partitioning the dataflow node execution across the client and server (i.e., execution plans). Then, these plans are encoded into feature vectors to train models to predict the most suitable plan. In this section, we first explain the challenges of developing cost models for visualization (Section 5.1), and we then describe the plan enumeration (Section 5.2) and ranking strategies (Section 5.3) implemented in the VegaPlus optimizer.

5.1 Visualization Optimization Challenges

There exist unique challenges for interactive visualizations. First, with VegaPlus's cross-stack setup, the optimizer needs to coordinate between the backend user-provided DBMS, Vega runtime and the data transferring cost. For example, although MonetDB, DuckDB and PostgreSQL all have baseline cost models and query optimization methods that work well for their target use cases [43, 45], they fail to provide adequate performance under real-time interactive visualization scenarios [2]. Cost models are difficult to tune, especially when agnostic of the chosen systems. However, our goal is to efficiently utilize available resources with any provided configuration. Furthermore, a ready-to-use cost model for the dataflow operations in Vega currently does not exist, making a fair performance comparison nearly infeasible. Second, visualization workloads are different from typical OLAP workloads. A visualization can generate different queries depending on which interactions the user performs in the visualization interface. Thus, the best plan that works for a static dashboard might not perform well in the long run due to shifts in the queries executed in response to user interactions.

More broadly, machine learning techiniques [29, 32, 33, 52, 61] have been explored to replace core components in database query optimizers for efficiency and accuracy. Traditionally, heuristic-based cost models are used to find the query plan with the minimum estimated cost. Cost estimations can be convenient to compare different plans of the same query. However, they cannot be directly used as an approximation of execution time without extensive tuning, and neither can they be taken out of the context to compare across different queries. Most learned optimizers aim to predict query plan latencies, but still suffer from high training costs and difficulties in model updating required by data or system configuration changes.

That being said, there are specific opportunities in visualization that we do not observe in more general optimization scenarios. For example, many Vega visualizations tend to involve the same combinations of data transformations, even across different visualization types. This reduces the number and diversity of scenarios required to develop effective optimization models. Furthermore, this constrained visualization design space may suggest that complex models such as deep learning models may not be needed to achieve good performance.

VegaPlus' Pairwise Approach. We propose comparator models that focus on the relative performance difference between a pair of plans. More recent work [64] sidesteps the issues with *pointwise* (i.e., predicting cost/execution time) by instead applying pairwise ranking of plans to identify better and worse plans. Similarly, [13] uses pairwise comparison for index tuning. In VegaPlus, we formulate the problem of comparing the execution cost of two visualization partitioning plans as a classification task. While traditional pointwise learned cost models learn to minimize the prediction error for each plan, a pairwise model learns to minimize the error of comparison [13, 64], which is intuitively more robust to the goal of plan selection in optimizers. With successive pairwise comparisons, we should arrive at a relatively efficient plan for a given Vega specification.

The pairwise approach provides benefits beyond a model for pairwise comparison. They produce insights to build rule-based heuristic models and cost models. The traditional ML algorithms are more transparent and easier to interpret than complex neural networks and deep learning models. For example, we can inspect the feature weights of a trained linear model to understand the feature

importance. In this case, heuristics and priority can be extracted and translated to heuristics to build a rule-based optimizer, or as a starting point towards an efficient cost model that adapts to data input, dashboard designs, user interactions, and system configurations. In Section 5.3, we elaborate on how we can synthesize baseline cost models as a by-product of training a comparison model using the linear RankSVM algorithm. We also explain our approach to mapping observations of feature importance in learned models (including both the RankSVM and the Random Forest models) into cost model heuristics.

5.2 Plan Enumeration

Given a Vega specification, VegaPlus's plan enumerator generates n candidate plans for executing this specification. Execution plans are different ways to partition the dataflow's transform operations across the Vega runtime running in the browser and the server-side (or remote) DBMS.

Data Dependency Checking. Transform operations are contained in the `data pipeline` component of the specification. The data pipeline transforms the raw data source into one or multiple smaller datasets that can be directly mapped to visual attributes such as x- and y-axes, color mappings, etc. To coordinate the computation, we first check for data dependencies to identify data results that must be preserved on the client, i.e., values explicitly referenced by the Vega runtime. Then, we enumerate candidate execution plans that satisfy the required data results and constraints.

In the original client-side Vega specification, the existence of the intermediate results is necessary for multiple reasons: a) other specification components, e.g., `scale`, `mark`, can refer to them, b) one intermediate result can be the source for multiple children data entries to avoid duplicate computation. VegaPlus may not need to maintain every data entry, except for the former reason. Thus, as the first step of plan enumeration, we traverse the data pipeline in the specification to identify these data entries, and derive a computation graph indicating data entry dependency and data results that must be preserved.

Enumerating the Data Pipeline. Given a specification, the enumerator outputs execution plan candidates indicating where each operator needs to be executed, on the server or on the client. In theory, a specification with n operators results in 2^n execution plans. However, in reality, there are fewer possible candidates due to the visualization dataflow search space.

The data movement in the dataflow graph is in a single direction because the data source resides on the DBMS and the visualization view is rendered on the client at the end. Considering any path in the DAG that starts from a data source root and ends with a leaf operator (whose output is directly mapped to visual components without further transformation), there should exist one and only one *split point* where the data flows from the server to the client. The split point is conceptual so that all operations that are upstream to the split point are executed on the server, and all that are downstream should be on the client. However, two paths may share the same prefix, and diverge right after an intermediate result. If the split points of both occur at the same position before the intermediate result, then we consider they share the same split point and the same server-side operations. Otherwise, the SQL queries for different paths are executed separately.

We enumerate all possible execution plans by traversing the computation graph and permuting the way an operator is executed. An operator can be rewritten in SQL only if the rewriting is supported and the parent to the data entry containing the operator is not reserved by the dependency-checking process. Meanwhile, we make sure that, for each operator-rewritten operator, all its ancestors are rewritten and wrapped as its subqueries. We consolidate paths only if they have the same prefix rewritten in SQL to avoid querying redundantly.

For the running example in Figure 3, the original bin operator is supposed to append the bin start and end values to each data item, while the aggregate operator groups and counts the items

in each bin. The example candidate plan split after all data pipeline nodes, assigning all operations to be executed on the server. The extent query is set aside because it outputs a signal requested by other operators. The aggregate operator absorbs and merges with the bin operator's query when query rewriting is performed alongside the enumeration process. In such a way, the plan avoids fetching the binning result, which has the same number of rows and two more columns, compared to the raw data source.

5.3 A Pair-Ranking Optimizer

VegaPlus optimizer adopts a *pairwise* approach to find the best plan. Given each pair of plan candidates, the comparator model ranks them by which one is more efficient.

5.3.1 Plan Encoding. To teach a model how to discern one plan from another, we need a way of encoding each plan into a vector format understood by existing modeling libraries. To achieve this, we vectorize each enumerated execution plan p to its feature vector v. A plan vector v is an array of features $v = \{f_1, ..., f_k\}$ that represent the characteristics of the dataflow graph compiled from the data pipeline in an execution plan. We extract v by traversing the dataflow, maintaining a counter for all operator types and the sums of output cardinality for each operator type. Since the cardinality can span a wide range, we apply a min-max normalizer for the cardinality features in the vector.

We omit the structural features in encoding because the operations in JavaScript/Vega are single-threaded and blocking. Within each single interaction, the operations are serialized without loops. The operator counts capture the distribution of operator types. A larger number of VDTs and fewer number of other client-side operations indicate a larger extent of query rewrite. The output cardinality of VDTs reflects both the SQL query cost and the network cost. These statistics may not be sufficient to derive a prediction for execution time; however, with a pairwise comparator, they are effective in distinguishing the difference between two candidates of the *same* specification.

5.3.2 Plan Comparators. Given a set of plan vectors for a target Vega specification, the optimizer's plan comparator must infer the best plan from the vector set. Specifically, for each visualization query Q, the plan enumerator outputs candidate plans $p_1, ..., p_n$. Then, the plan comparator picks the best plan $opt(p)$ by iteratively selecting the better plan of each pair (p_i, p_j) where $i < j$. We have implemented a suite of prediction models to make these pairwise comparisons. Specifically, given a pair of plans (p_i, p_j), each comparator model predicts whether v_i is preferred based on $sign(Compare(v_i, v_j))$.

We implement three types of comparator models:

Naive model. The naive model is learned through a pairwise comparison of encoded plans, with binary labels indicating which is the faster plan among the two. We use off-the-shelf ML packages to compare two classification methods – specifically, the linear RankSVM and Random Forest tree-based models.

We use a RankSVM (Support Vector Machine) model [21] to fit the difference of encoded vectors to the target label, so that the model, after training, would assign weights to features. At inference time, the cost of a visualization plan can be calculated as the weighted sum of the features with the weights extracted from the trained model. Hence, we do not need to invoke the model $n(n-1)/2$ times for every pair to find the best plan; rather, we calculate n cost values and return the one with the lowest cost. More formally, the model $Compare(v_i, v_j)$ is learned from a dataset where each entry $(v_i, v_j; y)$ is a plan vector pair associated with the label y indicating whether $latency(v_i) < latency(v_j)$ $(y = 1)$ or vice versa $(y = 0)$. Using the RankSVM algorithm, eventually, we train a cost model and receive a weight vector w for the features, where the class of a pair is determined by the sign of $Compare(v_i, v_j) = Cost(v_i) - Cost(v_j) = w^T(v_i - v_j)$. Given a set of all vectors $V = \{v_1, ..., v_n\}$,

linear regression is performed on the dataset $D = \{(v_i, v_j; y) | 1 \leq i \neq j \leq n\}$ of size $n(n-1)/2$ by minimizing the hinge loss with the optimal weights. The loss function is defined as:

$$L = \frac{1}{|D|} \sum_{d=(v_{d1}, v_{d2}; y_d) \in D} max(0, 1 - y_d w^T (v_{d1} - v_{d2}))$$

With RankSVM, we can formulate a cost function with the learned weights and return the plan with the lowest cost in linear time. The Random Forest (RF) model, on the other hand, does not generate weights for each feature via learning. In contrast, we need a wrapper around the RF model that assigns a vote for the better plan among a pair for each prediction, and then picks the best plan that receives the majority votes.

Heuristic model. Since there are too many visualization scenarios for manual cost model development, we choose to generate training data and use ML models to explore the design space with the naive models. Furthermore, we explore the possibility of reverse engineering useful findings to extract rules from trained models into heuristics and cost models. Based on the characteristics in the plans we observe and the weights derived in the naive model, we design a few simple rules as an alternative heuristic model. The intuition here is that if the naive model is learning useful information from our training data, it should be possible to summarize what the naive model has learned using simple rules, producing a streamlined and human-interpretable ranking model.

We implement a heuristic model with prioritized rules, where the prioritization is derived from the weights of the trained linear RankSVM model and the feature importance is calculated from RF. For example, the first rule states that a plan receives a vote if the sum of query results cardinality is smaller than the other plan by a factor of α. If the normalized difference is within the threshold, we break the tie by the next rule that prefers plans with more aggregations on the client-side. We follow the rules to compare a pair of plans until a decision has been made or all the rules have been applied.

Compared to the naive model, the heuristic model ranks a pair entirely based on plan characteristics without modeling their cost. We select the best plan by ranking every pair and finding the one with the most wins. Even though the heuristic model can be deployed directly without any offline training process, it costs more time to compare every pair.

Random model. As a sanity check, we also implement a random model that picks a random plan in a pair. As the number of viable plans grows, we expect the random model to perform worse than the other models, providing a useful performance baseline for comparison. We use the model in Section 7 (Evaluation) to assess the pairwise comparison and the end-to-end performance of the naive and heuristic models.

5.4 Consolidating Plan Decisions Across Interactions

When optimizing a static dashboard (i.e., without interactions), we have exactly one form of the dataflow graph to optimize, which we describe in the previous subsections. However, in the case of interactions, the queries to be executed may change depending on how interactions introduce new Vega transforms to the dataflow graph. As a result, the best plan for one interaction may not be the best plan for other interactions, even within the same Vega specification. To account for this variation, we collect separate plan vectors per interaction. As interactions are triggered and the corresponding signals update, the dataflow of each execution plan re-evaluates. This re-evaluation is partial, affecting only the subset of operators that depend on the updated signal. To handle this variation, we traverse the dataflow again to extract the vectors for the operators whose timestamps are current. For an exploration session s where the dashboard is updated t times, we obtain a set of t vectors $s_p = \{v_{p_0}, v_{p_1}, ..., v_{p_t}\}$, where v_{p_0} is the vector for initial rendering.

With the comparator model, each interaction episode nominates its own best choice. The chosen plans can be different based on interaction type, operators involved and interaction parameters. To consolidate the candidates and derive a final decision, we sum up the costs of all interactions for each plan candidate p and return the best plan with the minimum total cost by $\text{argmin}_p \sum_{v_{p_t} \in s_p} Cost(v_{p_t})$. As for the heuristic model, we replace the cost with the number of wins for each plan.

We also allow the weights to be configurable if certain episodes are considered more significant. For example, the user might downweight the interactions that are far in the future and prefer a plan that works best for the immediate next steps. Similarly, designers might also downweight the initial rendering, since users may be more lenient for initial rendering latency (i.e., cold start) if the interactions are more seamless [5].

5.5 Caching

Similar to prior work [1], We follow a straightforward caching scheme with two levels: *(a)* a client-side cache and *(b)* server-side middleware cache. The caches store the results of executing SQL queries executed on the DBMS. Each cache is implemented as an array of dictionary objects, where the key for each object is the SQL query string executed and the value is the result obtained for that particular query from the DBMS. Each cache has a fixed size in terms of total queries and follows the first-in-first-out replacement policy, with additional checks to avoid duplicate entries. When a query in the execution plan has to be executed, we check all possible sources in the order of the client's cache, server cache and finally the DBMS for full execution. To avoid the cached entity being too large, we set a threshold for the size of the query result. Only if the result size is below the threshold will we store it.

6 BENCHMARK DATA GENERATION

Inspired by recent exploration benchmarks for DBMSs [3, 15], we contribute a benchmark to evaluate the performance of Vega and VegaPlus. To build the benchmark, we collect five real-world datasets to act as visualization data sources and scale them to different sizes as done in prior work [3, 15]. In parallel, we curate seven Vega templates covering common visualization designs, which we describe in Section 6.1. Finally, in Section 6.2 we describe how we simulate user interactions for each template to generate interaction-driven query workloads for measuring system performance.

6.1 Generating Visualizations From Templates

To generate realistic training data and assess VegaPlus's expressivity, we collect seven visualization templates including two static charts, two single-view interactive charts, and three interactive dashboards. A unique feature of our benchmark is that all seven templates are independent of our source datasets, enabling us to evaluate any valid pairing of Vega code template and data source. We display the collection in Figure 5 with their native source dataset. Here we describe each template and explain how we parameterize them with the input datasets for training data generation.

Trellis Stacked Bar Chart. This is a multi-view chart, where each individual view is a stacked bar chart representing the cumulative sum of another categorical field, faceted by a third categorical field. Vega's `stack` and `aggregation` transforms are applied.

Line/Area Chart. This template applies a `timeunit` transform to the `x` channel field that bins the time-series data in different intervals of choice. The encoding can be simply changed to the `area` mark, which does not affect the rest of the dataflow processing.

Interactive Histogram. This template bins a quantitative field on the `x` channel and applies `aggregation` to count observed values from the `y` channel. Both the bin size and choice of the field

are parameterized, allowing them to be controlled by a slider and a drop-down selection menu, respectively.

Zoomable Heatmap. This chart uses 2D binning and aggregation. The panning and zooming interactions it supports trigger recalculation of binning and aggregation for density.

Crossfiltering With Three 2D Histograms. This dashboard contains three histograms linked by interactions. In each histogram view, a `brush` interaction filters and re-aggregates all linked views according to the brushed filter range.

Heatmap and Bar Chart. The heatmap counts observations binned along two categorical fields on the `x` and `y` channels. It is linked to a bar chart that counts the number of records for a third categorical field. By clicking on the bars in the bar chart, the heatmap filters and updates the density accordingly. An additional slider widget is provided to adjust the bin size in the heat map.

Overview+Detail Chart With Bar Chart. `Interval brushes` can be applied to the overview area chart to update how data points in the detail view are binned. Then, a bar chart groups the data by a categorical field and the user can filter both the overview and detail views by clicking on individual bars.

6.2 Parameter and Interaction Simulation

Parameters and interactions are generated in two steps. First, we populate the template to obtain a specification for initial rendering. The process is illustrated in Figure 4. Then, we simulate the interaction sequence and generate interaction parameters based on their type. Within each template, there are specific contents that need to be populated with data. For example, the template allows for the dynamic replacement of `data fields` in the specification. Given source data and the data type, we randomly pick from the suitable fields and fill in all the places that refer to this same field.

To evaluate VegaPlus's performance in static and interactive contexts, we implement a workload generator to simulate potential interactions. Here, workloads are sequences of interactions supported by the given dashboard template. In Vega, interactions are implemented using signals. Users can specify listeners on the desired interaction/visualization components, and listeners emit data (i.e., signals) reporting the current interaction state, which can be used to update the visualization(s). The signal types from our templates support interactions performed using separate widgets (e.g., slider filters and drop-down menu selection) as well as interactions performed directly on target visualizations (e.g., panning and zooming, brushing and linking, and selection by clicking).

For each template, we parse the specification to collect all possible signal types, which we use to generate individual interactions. We repeatedly call this interaction generation code to produce interaction sequences to form a workload. Given that data fields are selected randomly to populate our Vega visualization templates, we must gather statistics from the randomized dataset to determine the interaction/signal parameters. For example, calculating the corresponding data range for sliders and brush filters, and using this data to randomly select filter ranges for simulated interactions. To bind the input for drop-down menu selection, valid options need to be collected from the unique values in categorical data fields. Note that these probabilities are also configurable.

7 EVALUATION

To evaluate VegaPlus, we first analyze the customized workload in Section 7.2. Then, we examine how different models perform in the visualization initial rendering phase (Section 7.3), and the interaction sessions (Section 7.4). Finally, we compare its performance to Vega and VegaFusion as the baselines in Section 7.5.

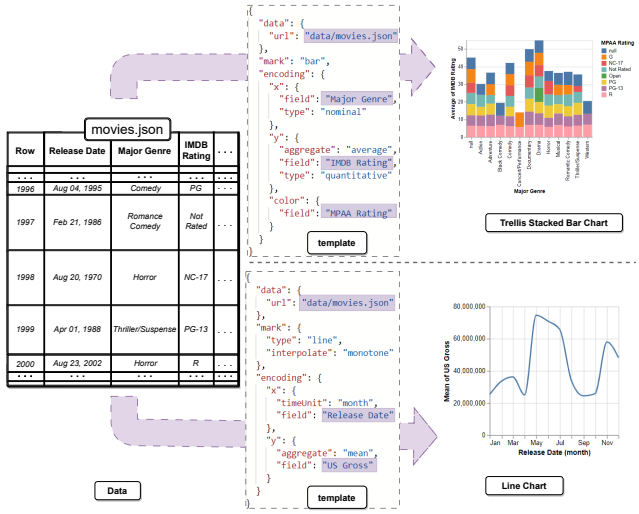


Fig. 4. Given a source dataset, specification templates are populated with values highlighted in purple. As a result, concrete visualizations can be rendered.

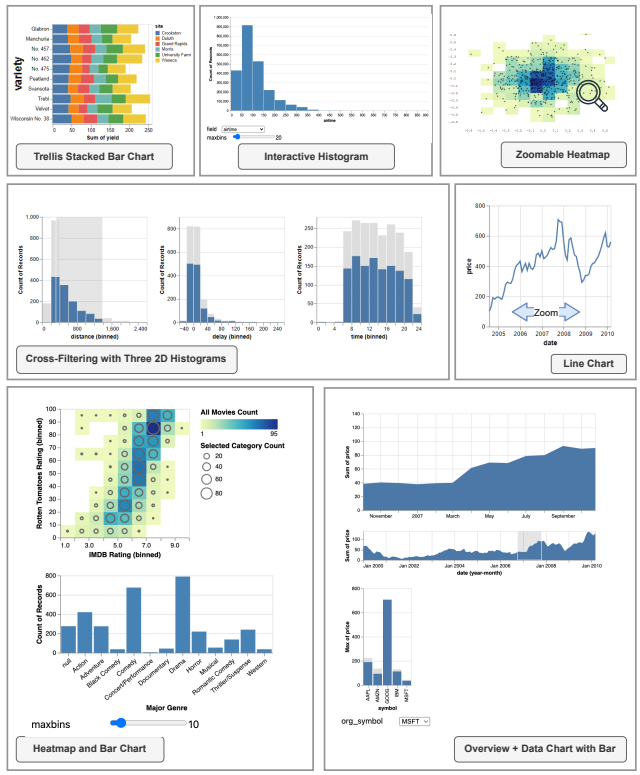


Fig. 5. Templates collected for evaluation.

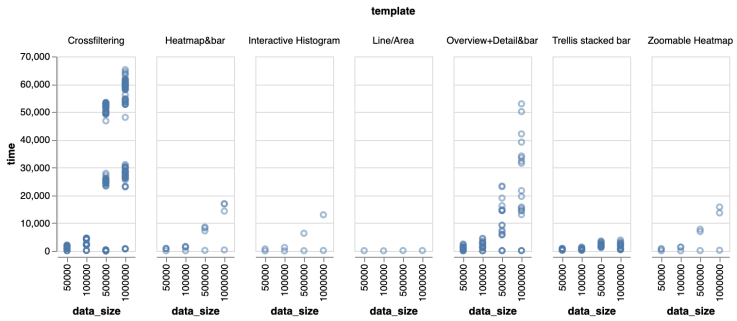


Fig. 6. Distribution of plan candidates' execution time of the initial rendering for each template.

7.1 Experiment Setup

Data. We evaluate the VegaPlus optimizer with a customized benchmark using the templates described in Section 6.1. For each template, we generate a workload of 10 sessions, each with 20 interactions. Given the template, each time we randomly pick one dataset as the source data and simulate the parameters and interactions. We vary the source data sizes from 50,000 rows to 1 million rows.

Setup. We deploy the VegaPlus system on a Linux machine with two 12-core 2.6 GHz Intel Xeon processors and 512GB RAM running Rocky Linux 9.1. The SQL queries are executed on PostgreSQL 15.1 with the default configuration. To compare with VegaFusion more fairly, we switch to DuckDB(v0.6.1) for both VegaPlus and VegaFusion as the SQL query execution engine.

7.2 Analysis of Templates and Their Plan Enumeration Space

Enumeration space. We report the number of operators and the number of plans enumerated for each template in Table 1. Templates for single-view visualizations have fewer operators than the multi-view dashboards, and enumerate fewer plan candidates as a result. We obtain more training data points in pairs by generating 10 sessions each with 20 interactions for each data size from 50,000 to 1 million rows. For each template, we also report the number of data points in the same table. For a static template with n candidates, we receive $10 * \binom{n}{2}$ pairs for each data size. For interactive templates, we obtain $200 * \binom{n}{2}$ pairs for each data size due to the interactions.

Our template collection is representative of how it covers the most common designs and data transformation types observed in the official Vega visualization gallery [1]. For most of our templates (i.e., the common case for Vega users), the combinatorial enumeration does not result in a prohibitively large training overhead. For example, training on hundreds of plans is completed within one minute, and making predictions with the random forest model takes tens of microseconds. Enumerating plans for model training takes less than one second, even for the template with the most plans (Crossfiltering With Three 2D Histograms). That being said, pruning strategies could be applied in the future to reduce the enumeration and plan ranking time. One strategy is to use simple heuristics to constrain the search space. For example, we can prune all plans involving enumerated operators with an estimated output cardinality above a certain threshold. Alternatively, we can apply bottom-up boundary pruning [25] to reduce the search space from $O(2^n)$ to $O(n)$, where n is the number of operators.

[1] https://vega.github.io/vega/examples/

templates	# of Operators	# of plans	# of pairs
Trellis Stacked Bar Chart	3	4	60
Line/Area Chart	2	3	600
Interactive Histogram	3	4	1200
Zoomable Heatmap	5	4	1200
Crossfiltering With Three 2D Histograms	14	111	1221000
Heatmap and Bar Chart	7	5	2000
Overview+Detail Chart With Bar Chart	8	19	34200

Table 1. Characteristics of each template, and the number of plan candidates and training data records generated by the enumerator.

Training data label space. We execute the queries in each template (and interaction sequence) to collect the execution times, which we use to label the pairs with the most effective plan. To help understand the distribution of candidate execution time for each template, we plot the data size on the x-axis and every execution time of the 10 sessions for each plan on the y-axis as a faceted scatter plot in Figure 6. Note that we only plot the initial rendering without the interactions because they are more easily affected by their parameters, which introduce more factors to the analysis. We observe that templates with a larger number of candidates caused by a more complex data pipeline can have a wide range of plan execution times. Further, the plans are more sensitive to the scale of the data source: as the size increases, the plans result in higher latency. For templates such as the "Crossfiltering with three 2D histograms" and "Overview+Detail chart With Bar Chart", there exists clusters of data points within each column of data size. And as the data size increases, the cluster margins become less distinguishable. An ideal optimizer should be able to learn these patterns.

In general, there are significantly more slow plans than fast plans when the enumeration space is large. A traditional pointwise model might suffer from this imbalanced plan enumeration space, given a lack of data for accurate prediction of plan execution times. However, with a pairwise model, we were able to switch from predicting execution times to the simpler problem of predicting the significance and signs of difference between plans. In the following subsection, we show that even when the comparator model has lower pairwise accuracy, it can output sufficiently fast plans due to its "learning-to-rank" nature.

Keeping all valid candidate plans in the training data preserves the original plan space, helps us observe what information is learned by the naive model, and allows us to implement the heuristic model based on the results. However, executing all valid plans to generate labels can be time-consuming. To reduce training data collecting time as future work, we can adopt an active learning approach [18] to identify a small set of representative plans to execute from which we can infer labels for non-executed plans [62].

7.3 Optimizer Models in Comparison For Template Initial Rendering

We compare the RankSVM-based model, the heuristic model and a model that randomly picks a plan in a pair, and evaluate their accuracy in predicting the better plan of each pair in Table 2. With the RankSVM model, we randomly split all collected data into a training set with 60% of the data points and the rest 40% for the test set.

models	accuracy for input sizes			
	50000	100000	500000	1000000
RankSVM	0.755	0.781	0.803	0.744
Random Forest	0.837	0.844	0.868	0.816
heuristic	0.709	0.711	0.710	0.715
random	0.500	0.500	0.499	0.501

Table 2. Model prediction accuracy for plan pair comparison.

models	predicted execution time (ms)			
	50000	100000	500000	1000000
RankSVM	112.77	182.22	**420.82**	890.53
Random Forest	**104.61**	**159.24**	433.04	**715.58**
heuristic	**104.61**	**159.24**	433.04	**715.58**
random	1480.95	3602.70	40902.96	45857.96
optimal	**104.61**	**159.24**	**420.82**	**715.58**

Table 3. Overall performance of models compared to ground truth (optimal plan).

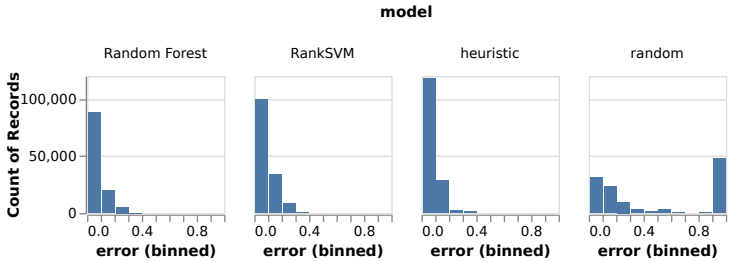


Fig. 7. Distribution of scaled errors for each model.

7.3.1 Accuracy and Performance. The Random Forest model has the highest accuracy when trained on data with varied input sizes. While the RankSVM is slightly less accurate, it does not generalize well across different input sizes. When the training and testing processes are performed on different data size, the accuracy decrease to between 0.60 and 0.65 (less than the heuristic model). The heuristic model does not require training and makes predictions based on the properties between two different plans. Thus, its accuracy relies on whether the rules capture the correct criteria of good plans as well as whether the rules are ordered to reflect how important they are. The random model has an accuracy of around 0.5 as expected, as it is predicting a winner between pairs of plans.

Initially, we assumed that higher accuracy in predicting between pairs of plans should translate to picking the fastest plans. However, our results suggest this is not the case. Table 3 reports the execution time of the plan picked by each model and compares them with the actual fastest execution time collected from an exhaustive search. The heuristic model achieves the best result that is the same as the Random Forest even though the model accuracy is slightly worse than the RankSVM-based model. These results naturally led us to question why accuracy does not seem to predict execution times, which we investigate in the next section.

7.3.2 Distribution of Errors. Next, we investigate how costly prediction errors were for each model. Specifically, we analyze all the incorrect predictions made by each model to see how far from optimal the prediction was in terms of execution time. To make the results comparable across conditions, we normalize the differences in execution time to fall between 0.0 and 1.0. We show the distribution of scaled errors for each model with input data size as 1 million rows in Figure 7. The scaled error is measured by the average of the ratio between the difference in the execution time of the picked plan P_i and the actual better plan A_i with less execution time.

$$ScaledError = \frac{1}{n}\sum_{i=1}^{n}\frac{|P_i - A_i|}{P_i} \tag{1}$$

Thus, the first bin represents the count of prediction errors for the pairs of plans with the closest execution times. Similarly, the last bin of each histogram represents the total prediction errors for the pairs of plans with the biggest difference in execution times.

First, we see that the execution time differences for all possible plans are roughly divided into clusters where the differences within are relatively small (near 0.0) and the differences between clusters are large (near 1.0). For example, the random model appears to have two error clusters in Figure 7 where it makes many low-cost mistakes (see the bins near 0.0) but also many high-cost mistakes (see the final 1.0 bin). This makes sense, since the random model uses a uniform distribution to select plans, and there are significantly more bad plans than good plans in the training dataset.

While the RankSVM model is more accurate in general, it makes more incorrect predictions for pairs with larger execution time differences compared to the heuristic model. In other words, when the RankSVM model is wrong, it tends to make costly prediction mistakes. In contrast, the heuristic model can better distinguish the pairs and make the correct prediction when there is a larger difference in execution times. We see this in Figure 7 where the first bin of the RankSVM error histogram is lower than the first bar for the heuristic error histogram; this means the heuristic model made more mistakes with the pairs of plans with the closest execution times compared to the RankSVM model. These findings suggest that it is not only important for a model to identify very fast plans but also to *avoid very slow plans* to yield consistent, end-to-end performance gains.

7.4 Optimizer Models in Comparison For Interactions

We compare the models' performance in interaction sessions. We generate all pairs of plans for each interaction episode as training and testing datasets.

Pairwise Prediction Accuracy. In Table 4, we report the pairwise prediction accuracy for the three models. The RankSVM model achieves higher accuracy compared to when it is trained only on the initial rendering data points since there is a larger number of pairs in the training dataset. On the contrary, the heuristic model with rules tailored for static dataflow generates significantly lower latency both compared to the RankSVM model and itself with only the initial rendering data points. Different from initial rendering data points, the interaction plans involve parameters of different selectivity and result in more variation for cardinality results in the encoded vectors. Further, the heuristic model makes decisions based on general rules such as "prefer the plan with fewer operator X" or "prefer plans with a smaller sum of cardinality", while the RankSVM model uses weights to reconcile more features.

Analysis of Plan Consolidation result. We evaluate the plan consolidation strategy and discuss the selected plan performance in terms of interaction sessions. The RankSVM model successfully selects the plans with minimum latency for all templates except for the "Overview+Detail Chart With Bar Chart", while the Random Forest model succeeds in this case. However, for this template,

models	accuracy for input sizes			
	50000	100000	500000	1000000
RankSVM	0.790	0.851	0.860	0.861
Random Forest	0.891	0.887	0.908	0.891
heuristic	0.621	0.494	0.576	0.566
random	0.499	0.501	0.499	0.500

Table 4. Model pairwise prediction accuracy with interaction episodes.

models	predicted execution time (ms)			
	50000	100000	500000	1000000
RankSVM	463.26	584.93	932.70	1695.19
Random Forest	442.37	584.93	932.70	1695.19
heuristic	8048.15	16540.91	90736.35	332776.94

Table 5. Average overall performance of models for template "Overview+Detail Chart With Bar Chart" per interaction session.

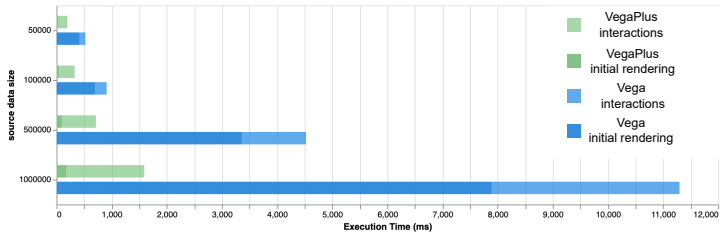


Fig. 8. Average execution time per session for Vega and VegaPlus with the RankSVM comparator model.

the RankSVM optimizer picks the second-fastest plan of all, and the fastest plan is ranked third by the comparator. First, this template generates the second most number of plan candidates, following the "Crossfiltering With Three 2D Histograms" template. Further, the template contains a unique `Timeunit` transform that bins the timestamp in the specified unit (e.g., month), which only exists in the workflows for this template. The results are shown in Table 5 in comparison with the heuristic model. The candidates nominated by each interaction are correlated to the combination of interaction type and parameter selectivity, suggesting future work on updating the model to adapt to dynamic workflows even though our results indicate high accuracy for fixed interaction sessions.

In Table 5, we report the average performance of heuristic model selected plans and compare it with the RankSVM and Random Forest results. The values are a summation of response time for each interaction per session, which does not include any think-time. We use the "Overview+Detail Chart With Bar Chart" template because it supports multiple interaction types and generates a relatively larger number of plan candidates. The heuristic model comparator consolidates decisions with the counts of nominations because the model only returns the rank of a pair without a score. As a result, the model favors the most frequent interaction types no matter how little they affect the overall session execution time.

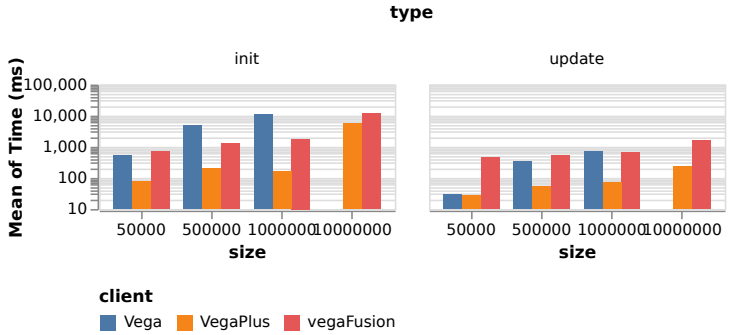


Fig. 9. Initial rendering and interactive update performance for Vega, VegaFusion, and VegaPlus (with the RankSVM comparator model).

7.5 End-to-end Performance Versus Baselines

We compare VegaPlus with the RankSVM comparator model against Vega, and report the results in Figure 8. Here we show the session average execution time for all interactive templates. In general, VegaPlus executes specifications faster than Vega for up to 86% with a data size of 1 million, and it scales better than Vega as the data size increases. Vega takes a significant amount of time to render the initial view, which includes loading the CSV file from the disk, constructing the dataflow graph and executing the initial view. The difference between Vega and VegaPlus in terms of executing iterations is smaller compared to the initial rendering. Interactions often involve a very small subset of the source data and trigger re-evaluation of a few operators in the whole data pipeline. Therefore, a large portion of interactions are not very time-consuming. In fact, we observe that, for smaller data sizes (i.e., 50,000 and 100,000), executing the interaction-only part takes a slightly longer time in VegaPlus than Vega. This is due to how the comparator consolidates the decision based on the session workflow. Many of the interactive templates are layered such that the data-intensive operations in initial rendering is a one-time effort. For example, the three histograms in the cross-filtering example (second row, first column in the Figure 5) require displaying the distribution of the whole source data in gray bars in the initial rendering. The operators attributed to the computation are never re-evaluated by any cross-filtering interactions. VegaPlus' RankSVM model is able to reason about the importance of episodes across a session to make the decision. It selects the candidate with faster initial rendering at the expense of a slightly slower interaction response.

We assess interactive performance with the Crossfiltering with three 2D histogram examples for Vega, VegaFusion [26] and VegaPlus. For VegaPlus, we choose the RankSVM comparator because it shows comparable end-to-end execution time and has a fast plan ranking time. We replace the backend data processor in both VegaPlus and VegaFusion with DuckDB. In Figure 9, VegaPlus results in better performance for both initial rendering and interactive updates even with smaller data sizes (i.e., 50,000 and 100,000) in contrast to using PostgreSQL. We extend the data size to 10 million rows with VegaFusion and VegaPlus, and drop the Vega condition because it cannot handle the data size. Query execution time also increases with datasets; hence the underperformance in VegaFusion and VegaPlus with larger data sizes. Future work may leverage indexing techniques [37] to reduce the latency in updating interactive linked views.

8 RELATED WORK

In this section, we discuss related work in visualization languages and optimizing for data exploration in big data environments.

8.1 Behavior-Driven Optimization

Data scientists often explore and analyze their data in predictable patterns [4, 5, 34, 41]. By modeling these patterns, DBMSs can anticipate and provision for user exploration actions through interaction-aware data caching [6, 8], pre-fetching [1, 10, 12, 35, 37, 41, 54], indexing [14, 16, 17, 44, 57, 58, 65], and partitioning [22, 31, 42, 51] techniques. However, these techniques are developed primarily for interactive analysis contexts where the interface design does not change, for example with a fixed dashboard design. Certain systems enable the creation of new visualizations, but restrict the types of supported visualizations (e.g., only bar charts [19]) or types of supported interactions (e.g., only pan/zoom interactions [57, 58]). We extend this work to support optimization in cases where the user (or the underlying system) can specify a wide range of new visualization and interaction designs.

8.2 Optimizing Visualization Languages

A number of projects aim to offer their users a language or API with which to design visualizations, where the data processing and rendering performance is optimized automatically by the language compiler/interpreter. Stolte et al. [56] present a table algebra for translating a core set of visualization designs into SQL queries and vice versa. Siddiqui et al. present a SQL-like language for efficient searching within and rendering of a set of possible visualization designs [53]. Tao et al. [58] present a specification language for constructing scalable pan-zoom visualizations, with a focus on scatterplot and heatmap visualizations. Ren et al. and Li et al. developed domain-specific languages to leverage WebGL and GPU accelerated data processing to enable fast interactive visualizations [27, 46] and machine learning [28] in browsers. In contrast, the Vega language enables many different visualization and interaction designs [49] but does not scale to support larger datasets [26].

More recently, Krutchen et al. [26] extend Vega by moving certain data transformations outside of the browser to a dedicated middleware layer written in Rust. Though effective, the range of possible visualization designs supported by these languages is limited. Furthermore, some solutions fail to match the scalability of DBMSs (e.g., [26]). Quansight open-sourced the `ibis-vega-transform` [24] Python library and Jupyter extension. It supports manually composing SQL expressions with Pandas-like API (i.e., ibis) to replace Vega transform pipelines and evaluating them on HeavyDB, which is optimized through hardware-accelerated parallelization. In this paper, we present an optimization approach that integrates the dataflow structure of Vega with new and existing database management techniques. Although VegaPlus shares the same idea of offloading operations, it automates query composing and plan selection. And it supports any user-provided backends including HeavyDB.

8.3 ML/Deep Learning for Query Optimization

Visualization dataflows share structural similarities with DBMS query plans and therefore can potentially be optimized with similar techniques. Learning-based methods have been applied to several vital components of query optimization such as cardinality estimation [29], cost models [52] and query performance prediction [33, 61]. Recently, ML has been used to tackle the optimization problem in an end-to-end fashion. Neo [32] uses deep learning to discover query plans directly by predicting the best possible latency. While the performance of ML models is generally superior to that of traditional ones, the learned query optimizers still suffer from some problems such as unstable performance, high learning cost, and slow model updating [64]. To combat these issues,

Lero [64] determines the relative order of plans rather than predicting the cost or latency to improve query optimization. We extend these ideas to a visualization dataflow context. Instead of learning query execution costs or latencies separately and comparing them later, our models directly solve the client-server partitioning problem using end-to-end plan enumeration and ranking strategies tailored to interactive data exploration environments.

9 CONCLUSION

VegaPlus is a system for automatically optimizing cross-stack visualization execution to support interactive exploration of large datasets. VegaPlus dynamically offloads computationally-intensive operations from the Vega runtime on the client side to a back-end DBMS. The interaction-aware optimizer adopts a pairwise approach to compare execution plans that partition the computation, and consolidate decisions across user interactions. We evaluate the performance and expressiveness of VegaPlus through benchmark experiments, which show improvements in scalability and interactive performance compared to the web-based Vega library.

10 ACKNOWLEDGMENTS

The authors wish to thank colleagues in the UW Interactive Data Lab and the UW Database Lab. This work was supported in part by the NSF through award number IIS-2141506.

REFERENCES

[1] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1363–1375. https://doi.org/10.1145/2882903.2882919

[2] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. 2020. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1571–1587. https://doi.org/10.1145/3318464.3389732

[3] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. 2020. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1571–1587. https://doi.org/10.1145/3318464.3389732

[4] Leilani Battle and Jeffrey Heer. 2019. Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Computer Graphics Forum* 38, 3 (2019), 145–159. https://doi.org/10.1111/cgf.13678 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13678.

[5] Leilani Battle and Carlos Scheidegger. 2020. A Structured Review of Data Management Technology for Interactive Visualization and Analysis. *IEEE Transactions on Visualization and Computer Graphics* (2020).

[6] L. Bavoil, S.P. Callahan, P.J. Crossno, J. Freire, C.E. Scheidegger, C.T. Silva, and H.T. Vo. 2005. VisTrails: enabling interactive multiple-view visualizations. In *VIS 05. IEEE Visualization, 2005.* 135–142. https://doi.org/10.1109/VISUAL.2005.1532788 ISSN: null.

[7] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[8] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06)*. Association for Computing Machinery, Chicago, IL, USA, 745–747. https://doi.org/10.1145/1142473.1142574

[9] Mackinlay Card. 1999. *Readings in information visualization: using vision to think*. Morgan Kaufmann.

[10] Sye-Min Chan, Ling Xiao, John Gerth, and Pat Hanrahan. 2008. Maintaining interactivity while exploring massive time series. In *2008 IEEE Symposium on Visual Analytics Science and Technology*. 59–66. https://doi.org/10.1109/VAST.2008.4677357 ISSN: null.

[11] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: continuous dataflow processing.

In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*. Association for Computing Machinery, San Diego, California, 668. https://doi.org/10.1145/872757.872857

[12] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: an automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, Snowbird, Utah, USA, 517–528. https://doi.org/10.1145/2588555.2610523

[13] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1241–1258. https://doi.org/10.1145/3299869.3324957

[14] Harish Doraiswamy, Huy T. Vo, Cláudio T. Silva, and Juliana Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1086–1097. https://doi.org/10.1109/ICDE.2016.7498315 ISSN: null.

[15] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. IDEBench: A Benchmark for Interactive Data Exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1555–1569. https://doi.org/10.1145/3318464.3380574

[16] Ahmed Eldawy, Mohamed F. Mokbel, Saif Alharthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. 2015. SHAHED: A MapReduce-based system for querying and visualizing spatio-temporal satellite data. In *2015 IEEE 31st International Conference on Data Engineering*. 1585–1596. https://doi.org/10.1109/ICDE.2015.7113427 ISSN: 2375-026X.

[17] Nivan Ferreira, Jorge Poco, Huy T. Vo, Juliana Freire, and Cláudio T. Silva. 2013. Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (Dec. 2013), 2149–2158. https://doi.org/10.1109/TVCG.2013.226

[18] Yifan Fu, Xingquan Zhu, and Bin Li. 2012. A survey on instance selection for active learning. *Knowledge and Information Systems* 35 (2012), 249–283. https://api.semanticscholar.org/CorpusID:5009954

[19] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2017. Revisiting Reuse for Approximate Query Processing. *Proc. VLDB Endow.* 10, 10 (June 2017), 1142–1153. https://doi.org/10.14778/3115404.3115418

[20] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[21] R. Herbrich, T. Graepel, and K. Obermayer. 1999. Support vector learning for ordinal regression. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, Vol. 1. 97–102 vol.1. https://doi.org/10.1049/cp:19991091

[22] Stratos Idreos, Martin Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*, Vol. 7. 68–78.

[23] Plotly Technologies Inc. 2015. Collaborative data science. https://plot.ly. https://plot.ly

[24] Quansight Inc. [n. d.]. ibis-vega-transform. https://github.com/Quansight/ibis-vega-transform

[25] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Bertty Contreras-Rojas, Rodrigo Pardo-Meza, Anis Troudi, and Sanjay Chawla. 2020. ML-based Cross-Platform Query Optimization. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1489–1500. https://doi.org/10.1109/ICDE48307.2020.00132

[26] Nicolas Kruchten, Jon Mease, and Dominik Moritz. 2022. VegaFusion: Automatic Server-Side Scaling for Interactive Vega Visualizations. In *2022 IEEE Visualization and Visual Analytics (VIS)*. 11–15. https://doi.org/10.1109/VIS54862.2022.00011

[27] Jianping Kelvin Li and Kwan-Liu Ma. 2020. P4: Portable Parallel Processing Pipelines for Interactive Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 26, 3 (March 2020), 1548–1561. https://doi.org/10.1109/TVCG.2018.2871139 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[28] Jianping Kelvin Li and Kwan-Liu Ma. 2020. P6: A Declarative Language for Integrating Machine Learning in Visual Analytics. *arXiv:2009.01399 [cs]* (Sept. 2020). http://arxiv.org/abs/2009.01399 arXiv: 2009.01399.

[29] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering* (Markham, Canada) *(CASCON '15)*. IBM Corp., USA, 53–59.

[30] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. 2013. imMens: Real-time visual querying of big data. In *Computer graphics forum*, Vol. 32. Wiley Online Library, 421–430.

[31] Mohammad Sultan Mahmud, Joshua Zhexue Huang, Salman Salloum, Tamer Z. Emara, and Kuanishbay Sadatdiynov. 2020. A survey of data partitioning and sampling methods to support big data analysis. *Big Data Mining and Analytics* 3, 2 (June 2020), 85–101. https://doi.org/10.26599/BDMA.2019.9020015 Conference Name: Big Data Mining and Analytics.

[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo. *Proceedings of the VLDB Endowment* 12, 11 (jul 2019), 1705–1718. https://doi.org/10.14778/

3342263.3342644
[33] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646
[34] Ben McCamish, Vahid Ghadakchi, Arash Termehchy, Behrouz Touri, and Liang Huang. 2018. The Data Interaction Game. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, Houston, TX, USA, 83–98. https://doi.org/10.1145/3183713.3196899
[35] Haneen Mohammed. 2020. Continuous Prefetch for Interactive Data Applications. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2841–2843. https://doi.org/10.1145/3318464.3384405
[36] Dominik Moritz, Jeff Heer, and Bill Howe. 2015. Dynamic Client-Server Optimization for Scalable Interactive Visualization on the Web. In *Workshop on Data Systems for Interactive Analysis (DSIA) at IEEE VIS 2015* (Chicago, IL).
[37] Dominik Moritz, Bill Howe, and Jeffrey Heer. 2019. Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, Glasgow, Scotland Uk, 1–11. https://doi.org/10.1145/3290605.3300924
[38] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738
[39] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (Sept. 2016), 75–83. https://doi.org/10.1145/2983551
[40] Bureau of Transportation Statistics. [n. d.]. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2010-09-30.
[41] Alvitta Ottley, Roman Garnett, and Ran Wan. 2019. Follow The Clicks: Learning and Anticipating Mouse Interactions During Exploratory Data Analysis. *Computer Graphics Forum* 38, 3 (2019), 41–52. https://doi.org/10.1111/cgf.13670
[42] Mirjana Pavlovic, Eleni Tzirita Zacharatou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2016. Space odyssey: efficient exploration of scientific data. In *Proceedings of the Third International Workshop on Exploratory Search in Databases and the Web (ExploreDB '16)*. Association for Computing Machinery, New York, NY, USA, 12–18. https://doi.org/10.1145/2948674.2948677
[43] PostgreSQL. 2019. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/.
[44] Fotis Psallidas and Eugene Wu. 2018. Smoke: fine-grained lineage at interactive speed. *Proceedings of the VLDB Endowment* 11, 6 (Feb. 2018), 719–732. https://doi.org/10.14778/3199517.3199522
[45] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212
[46] Donghao Ren, Bongshin Lee, and Tobias Höllerer. 2017. Stardust: Accessible and Transparent GPU Support for Information Visualization Rendering. *Computer Graphics Forum* 36, 3 (2017), 179–188. https://doi.org/10.1111/cgf.13178
[47] Neal Richardson, Ian Cook, Nic Crane, Dewey Dunnington, Romain François, Jonathan Keane, Dragoș Moldovan-Grünfeld, Jeroen Ooms, and Apache Arrow. 2023. *arrow: Integration to 'Apache' 'Arrow'*. https://github.com/apache/arrow/, https://arrow.apache.org/docs/r/.
[48] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030 Conference Name: IEEE Transactions on Visualization and Computer Graphics.
[49] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 659–668. https://doi.org/10.1109/TVCG.2015.2467091 Conference Name: IEEE Transactions on Visualization and Computer Graphics.
[50] W.J. Schroeder, K.M. Martin, and W.E. Lorensen. 1996. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of Seventh Annual IEEE Visualization (1996)*.
[51] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3127479.3131613
[52] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. *CoRR* abs/2002.12393 (2020). arXiv:2002.12393 https://arxiv.org/abs/2002.12393
[53] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. *Proceedings of the VLDB Endowment* 10, 4 (Nov.

2016), 457–468. https://doi.org/10.14778/3025111.3025126

[54] Manish Singh, Arnab Nandi, and H. V. Jagadish. 2012. Skimmer: rapid scrolling of relational query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 181–192. https://doi.org/10.1145/2213836.2213858

[55] Observable standard library. [n. d.]. https://github.com/observablehq/stdlib.

[56] C. Stolte, D. Tang, and P. Hanrahan. 2002. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (Jan. 2002), 52–65. https://doi.org/10.1109/2945.981851 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[57] Wenbo Tao, Xiaoyu Liu, Çağatay Demiralp, Remco Chang, and Michael Stonebraker. 2019. Kyrix: Interactive Visual Data Exploration at Scale. *arXiv:1905.04638 [cs]* (May 2019). http://arxiv.org/abs/1905.04638 arXiv: 1905.04638.

[58] Wenbo Tao, Xiaoyu Liu, Yedi Wang, Leilani Battle, Çağatay Demiralp, Remco Chang, and Michael Stonebraker. 2019. Kyrix: Interactive Pan/Zoom Visualizations at Scale. *Computer Graphics Forum* 38, 3 (2019), 529–540. https://doi.org/10.1111/cgf.13708 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13708.

[59] Pawel Terlecki, Fei Xu, Marianne Shaw, Valeri Kim, and Richard Wesley. 2015. On Improving User Response Times in Tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1695–1706. https://doi.org/10.1145/2723372.2742799

[60] Edward R Tufte. 1985. The visual display of quantitative information. *The Journal for Healthcare Quality (JHQ)* 7, 3 (1985), 15.

[61] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[62] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. *Proceedings of the 2021 International Conference on Management of Data* (2021). https://api.semanticscholar.org/CorpusID:235473953

[63] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *ACM Human Factors in Computing Systems (CHI)*. http://idl.cs.washington.edu/papers/voyager2

[64] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer.

[65] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, Snowbird, Utah, USA, 1555–1566. https://doi.org/10.1145/2588555.2610498

Received July 2023; revised October 2023; accepted November 2023