RESEARCH-ARTICLE

Physical Visualization Design: Decoupling Interface and System Design

YIRU CHEN, Columbia University, New York, NY, United States

XUPENG LI, Columbia University, New York, NY, United States

JEFFREY TAO, University of Pennsylvania, Philadelphia, PA, United States

LANA RAMJIT, Cornell Tech, New York, NY, United States

SUBRATA MITRA, ADOBE Systems India Private Limited, Noida, UP, India

JAVAD GHADERI, Columbia University, New York, NY, United States

View all

Proceedings of the ACM on Management of Data, Volume 3, Issue 3 (June 2025)
https://doi.org/10.1145/3725334
EISSN: 2836-6573

Physical Visualization Design: Decoupling Interface and System Design

YIRU CHEN*, Columbia University, USA
XUPENG LI†, Columbia University, USA
JEFFREY TAO, University of Pennsylvania, USA
LANA RAMJIT, Cornell Tech, USA
SUBRATA MITRA, Adobe Research, India
JAVAD GHADERI, Columbia University, USA
RAVI NETRAVALI, Princeton University, USA
ADITYA PARAMESWARAN, University of California, Berkeley, USA
DAN RUBENSTEIN, Columbia University, USA
EUGENE WU, Columbia University, USA

Interactive visualization interfaces enable users to efficiently explore, analyze, and make sense of their datasets. However, as data grows in size, it becomes increasingly challenging to build data interfaces that meet the interface designer's desired latency expectations and resource constraints. Cloud DBMSs, while optimized for scalability, often fail to meet latency expectations, necessitating complex, bespoke query execution and optimization techniques for data interfaces. This involves manually navigating a huge optimization space that is sensitive to interface design and resource constraints, such as client vs server data and compute placement, choosing which computations are done offline vs online, and selecting from a large library of visualization-optimized data structures.

This paper advocates for a Physical Visualization Design (PVD) tool that decouples interface design from system design to provide design independence. Given an interfaces underlying data flow, interactions with latency expectations, and resource constraints, PVD checks if the interface is feasible and, if so, proposes and instantiates a middleware architecture spanning the client, server, and cloud DBMS that meets the expectations.

To this end, this paper presents Jade, the first prototype PVD tool that enables design independence. Jade proposes an intermediate representation called Diffplans to represent the data flows, develops cost estimation models that trade off between latency guarantees and plan feasibility, and implements an optimization framework to search for the middleware architecture that meets the guarantees. We evaluate Jade on six representative data interfaces as compared to Mosaic and Azure SQL database. We find Jade supports a wider range of interfaces, makes better use of available resources, and can meet a wider range of data, latency, and resource conditions.

*Yiru Chen is currently at Adobe.
†Xupeng Li is currently at Databricks.

Authors' Contact Information: Yiru Chen, yiru.chen@columbia.edu, Columbia University, USA; Xupeng Li, xupeng.li@columbia.edu, Columbia University, USA; Jeffrey Tao, jefftao@seas.upenn.edu, University of Pennsylvania, USA; Lana Ramjit, lana.ramjit@cornell.edu, Cornell Tech, USA; Subrata Mitra, sumitra@adobe.com, Adobe Research, India; Javad Ghaderi, jghaderi@columbia.edu, Columbia University, USA; Ravi Netravali, rnetravali@cs.princeton.edu, Princeton University, USA; Aditya Parameswaran, adityagp@berkeley.edu, University of California, Berkeley, USA; Dan Rubenstein, dsr100@columbia.edu, Columbia University, USA; Eugene Wu, ew2493@columbia.edu, Columbia University, USA.

ACM 2836-6573/2025/6-ART197
https://doi.org/10.1145/3725334

CCS Concepts: • **Information systems** → *Data management systems*; *Application servers*; *Data structures*; **Database query processing**; **Query optimization**; **Query operators**; **Query planning**; • **Human-centered computing** → *Visualization*; **Visualization systems and tools**; **Interactive systems and tools**.

Additional Key Words and Phrases: data analytics, scalable visualization, visualization system optimization

ACM Reference Format:
Yiru Chen, Xupeng Li, Jeffrey Tao, Lana Ramjit, Subrata Mitra, Javad Ghaderi, Ravi Netravali, Aditya Parameswaran, Dan Rubenstein, and Eugene Wu. 2025. Physical Visualization Design: Decoupling Interface and System Design . *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 197 (June 2025), 27 pages. https://doi.org/10.1145/3725334

1 Introduction

Users rely on interactive visualization interfaces (data interfaces for short) to rapidly analyze, explore, and make sense of data. If the dataset is small (say, a few hundred records), the client can simply load the dataset into an embedded database like DuckDB [55] which can execute the corresponding SQL queries in milliseconds.

Unfortunately, as data continues to grow, it is typically stored in cloud DBMSs, which makes it increasingly difficult to build data interfaces that meet interface designers' latency expectations and resource constraints. A major reason is that cloud DBMSs are optimized for scale and not latency—communicating with the cloud DBMS at all is too slow for low-latency interactions—and so building a data interface effectively requires developing bespoke query execution optimizations and logic outside of the cloud DBMS. This involves manually navigating a huge optimization space that is sensitive to the interface design and resource constraints, such as client vs server data and compute placement, choosing which computations are done offline vs online, and selecting from a large library of visualization-optimized data structures [16]. The entire process requires interface design, backend development, and system architecture design, which involves a significant amount of work. These tasks are typically delegated to three distinct roles: interface designers, backend developers, and system architects. However, this approach often incurs substantial communication overhead and effort, especially since the interface designers may frequently adjust the interface design to accommodate evolving user needs. On the other hand, assigning all of these responsibilities to an individual, say the designer, would be overly demanding. In short, there is a lack of *Design Independence* to insulate data interface design and requirements from how the underlying system needs to be designed and optimized to support it. Design Independence allows interface designers to rapidly iterate without being bottlenecked by checking about with developers and architects about feasibility. Below shows an example:

Example 1. *Figure 1 visualizes vote counts for different members of congress. Suppose the dataset has grown over time and is stored in a cloud DBMS. The designer feels the DBMS latency is too high, and embarks on implementing a client-server system to reduce interaction latencies while trying three possible interaction designs.*

Figure 1(a) lets users click on one of two decades, and can be optimized by precomputation e.g., per decade statistics. Figure 1(b) lets users choose a date range. It may be infeasible to pre-compute the results for the quadratic number of queries, but if the statistic is distributive, a cumulative data tile is an optimization that can be compactly stored [26]. Figure 1(c) additionally lets the user choose the chamber in congress, but now requires computing a separate data tile for each chamber. If the client has sufficient resources, caching all or portions of these data structures on the client is another optimization choice; if the designer wants all of the interactions to respond within a few milliseconds, then client-caching may be necessary.

Although these interfaces in example 1 are similar, the appropriate optimizations and system design for each interface greatly vary due to an interlocking set of requirements and constraints:

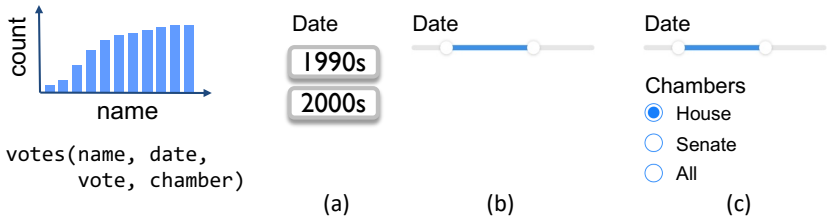

Fig. 1. Three variations of a data interface to analyze vote counts by congressional members. Users can choose (a) the decade, (b) a date range, or (c) a date range and chamber. Each design requires a different set of optimizations.

the queries triggered by user interactions, interaction latency expectations, database size, system architecture, and available resources. For instance, changing the interaction from buttons to a slider required changing the system architecture to pre-compute and use a visualization-optimized implementation of a data cube [50] instead of pre-computing a query subplan. Similarly, relaxing the latency requirements so each interaction can take several seconds would simplify the architecture because each query can simply run on the source database.

This complexity is challenging for a designer to manage. Given a prospective interface design, it is hard for a designer to even answer basic questions such as: *how do I implement this interface? Will it meet my latency expectations? And stay within the resource budget? What if the dataset grows 10× or my resources change?*

Unfortunately, no tools exist to answer these questions. Physical database design (PDD) tools (e.g., AutoAdmin [**?**], Azure SQL Database [47]) recommend indexes and materialized views to speed up a query workload and do take resource constraints into account; however, they do not reason about interactions nor provide latency guarantees. PDD is fundamentally designed for internal DBMS optimizations, whereas visualizations are sensitive to even milliseconds of latency and rely on bespoke data structures that are each optimized for specific query patterns and interaction types.

Thus, designers rely on visualization frameworks that are each designed around a specific visualization optimization. For instance, Kyrix [60] is designed for pan-zoom interfaces by leveraging PostgreSQL indexes, Falcon [50] specializes in data tiles for cross-filtering, VegaPlus [63] pushes Vega transformations to a database and caches the database results, and Mosaic [38] runs DuckDB in the browser or server to pre-aggregate group-by queries, while Cube.js [30] is designed around data cubes. Unfortunately, each framework constrains the designer to designs that stay within its optimization's narrow sweet spot.

We believe the dearth of tools is due to a confluence of reasons. First, there lack good representations of the interface's data flow, nor are there ways to express interaction latency constraints. Second, existing physical design tools, along with visualization frameworks, explicitly avoid modeling latency in terms of wall-clock time, making them unsuitable for interface design. Third, each visualization framework only supports one specific optimization. Fourth, these physical design tools are designed to reside within the DBMS, yet for data interfaces, *the cloud DBMS is the slow path and simply speeding up cloud DBMSs is not enough.*

For this reason, we advocate for a **Physical Visualization Design (PVD)** tool that decouples interface design from system design to provide *Design Independence*. Given an interface's underlying data flow, interactions with latency expectations, and resource constraints, PVD checks if the interface is feasible and, if so, proposes and instantiates a middleware architecture spanning the client, server, and cloud DBMS that meets the expectations. If infeasible, PVD should find a minimal relaxation to the resource requirements to meet the desired latencies. The tool should also be extensible to new visualization-specific data structures and solve the optimization problem quickly

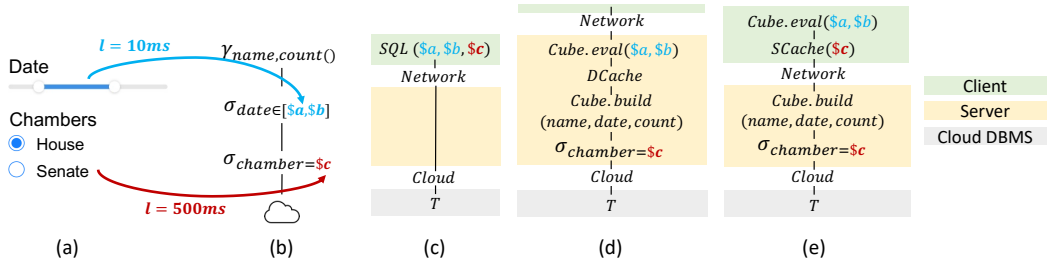

Fig. 2. (a) Interactions in running example, (b) logical DIFFPLAN, and execution plans that (c) send the query to the cloud DBMS, (d) pre-compute and evaluate interactions using cubes on the server, and (e) caches cubes on the client.

enough to support iterative interface design. Such a tool can be used to verify interface feasibility, plan for data growth, and quickly iterate on design choices while understanding the systems and resource implications.

Intuitively, PVD should not be possible because it makes stronger guarantees than physical database design (PDD)—widely considered difficult. PDD only attempts to reduce expected workload latency due to challenges of cost estimation, while PVD seeks to guarantee the wall-clock latency from when the interaction's query is issued to when results are available to be rendered.
In addition, there does not exist a representation of an interface's data flow that is useful for expressing data interfaces yet amenable to optimization—PDD query workloads are retrospective and do not exist when designing a new interface. Parameterized queries are too limiting to literal values: even modestly complex visualizations need to change query structures like expressions, grouping attributes, and potentially subqueries. Further, it is non-trivial to search the combinatorial space of physical plans while supporting bespoke data structures, cache placement policies, and split execution Choosing the best data structures and plans that meet constraints is also an NP-hard problem. Finally, we wish end-to-end optimization latency to be within a few seconds.

To this end, this paper presents the first prototype PVD tool called JADE. Inspired by recent work [23], JADE models the interface as a set of structurally parameterized query plans called DIFFPLANS (Section 3). In addition to parameterized literals supported by SQL parameterized queries, DIFFPLANS can parameterize attributes, expressions, and operators; we call each parameter a *Choice*. This provides a greater level of expressiveness while still remaining amenable to analysis. Interface interactions and widgets are responsible for binding these choices to their values in response to user interactions. This abstraction makes it easy for designers to specify per-interaction latency constraints while allowing JADE to analyze how those interactions translate to queries, e.g., the slider should update the bar chart within 20 ms.

EXAMPLE 2. *Figure 2(b) presents a simplified version of the DIFFPLAN for Figure 1(c). The DIFFPLAN uses Choice expressions to parameterize the range of the* `date` *predicate and the chamber of congress, followed by a count aggregation. Each arrow corresponds to an interaction labeled with its expected latency: the date slider should take no more than 10 ms, while choosing the chamber can take up to 500 ms. When the user interacts with a widget, it updates a set of parameter bindings, a dictionary mapping parameters to their bound values, which are sent to the physical plan.*

Given a set of DIFFPLANS, constraints, and database statistics, JADE extends rule-based optimization to find physical DIFFPLANS composed of relational operators, along with operators for custom data structures, network communication, and caching policies. JADE finds the best combination of physical plans that minimize expected interaction latencies and/or resource usage by encoding the problem as an integer programming problem (IP).

Example 3. *Figure 2(c-e) are three physical plans Jade may consider when optimizing Figure 2(b). Figure 2(c) is a SQL operator that uses the parameter bindings* `$a,$b,$c` *to generate and send a query to the cloud DBMS. It requires no client and server memory, but is slow. Figure 2(d) scans table T on the cloud DBMS, filters by chamber on the server, and builds a data cube on name and date. The cube is stored in a dynamic cache* (`DCache`) *on the server that is sized to store one data structure. The cube will be replaced if the chamber changes, but will be reused if the date changes. It uses no client memory and minimal server memory. Figure 2(e) differs in two subtle but important ways: the cache is static* (`SCache`)*, meaning it enumerates all parameters below it and caches all resulting subplans i.e., cubes for both chambers. The cache is on the client to avoid network latency. It requires memory on the client but offloads data structure construction to the server.*

Jade uses per-operator cost models to estimate interaction latency and provides knobs to trade off between conservative estimates, which guarantee latency constraints at the expense of fewer feasible plans, and aggressive estimates, which may sometimes violate latency constraints but are more likely to find feasible plans. Furthermore, Jade instantiates the physical plans and data structures as a middleware that accelerates the data interface.

In summary, we make the following contributions:

- We design the first PVD system called Jade that enables *Design Independence*, so the designer can focus on the interface and not on how the underlying infrastructure is architected;
- We propose Diffplans to model the data-flows affected by interactions in a way that is amenable to analysis and optimization;
- We formalize PVD as a cost-based query optimization problem that considers visualization-optimized data structures, client vs. server data and compute placement, and cache policies. We leverage existing rule-based optimizer design and solve the problem by combining rule-based plan enumeration with effective pruning heuristics and integer programming;
- We develop cost estimation models that enable designers to trade off between latency guarantees and plan feasibility;
- Surprisingly, we find that low latency constraints (e.g., milliseconds to seconds) *simplify* the optimization problem by invalidating plans with unbounded intermediate result sizes, including joins whose fan-outs are not bounded [12]. Intuitively, unless a join is guaranteed to be fast (i.e., its result is small), it cannot be executed in response to a user interaction. This lets Jade sidestep the hardest part of query optimization: join ordering.
- We evaluate Jade on six representative data interfaces as compared to a state-of-the-art visualization framework Mosaic [38] and a commercial PDD tool [47]. We find that Jade supports a wider range of interfaces than Mosaic, successfully meets latency guarantees, makes better use of available resources, and can meet a wider range of data, latency, and resource conditions. Only Jade can meet the corresponding guarantees for a database of 100M rows with only 100MB client and 1GB server memory.

Scope. The interfaces in this paper refer to predefined dashboard-like interfaces, such as BI dashboards or data journalism visualizations [5, 33], which are not open-ended like Jupyter Notebook [1] or Hex [7] but still support a space of analyses that is difficult to design and optimize. The interfaces are not constrained by query complexity, as many other tools (such as Tableau [6], Metabase [2], and Retool [3]) are, and can leverage an extensible library of data structures to optimize new structural patterns in the analysis queries.

This paper focuses on defining an interface representation that is amenable to analysis, and an optimization framework to support highly interactive data interfaces. Although we sketch an API for designers to specify the inputs to Jade, we leave the development of a comprehensive library

to future work. JADE does not consider optimizations based on approximation because it changes the semantics of the interface and is confusing to users [54], and focuses on read-only analytics interfaces. The optimizations are external to the cloud DBMS.

2 Use Case and System Overview

This section introduces an end-to-end use case as a way of introducing the system.

2.1 Use Case

We describe how a designer would use JADE to design, optimize, and deploy the data interface in Figure 1. By interface design, we refer to the queries the interface can generate and the latency bound designers expect for each interaction. While our focus is the optimization framework, we sketch a dataframe-like API and how it is used to specify and deploy the interface.

The designer first uses the JADE library to define the DIFFPLAN and interactions that populate the bar chart. She defines three value-based choice variables, which can take values from their associated attribute domains, e.g., `c` can be any chamber, `dmin` can be any value of `date`. JADE supports many choice types including numeric ranges, subsets, optional, enumeration of expressions, table and attribute names, and operator subplans (Section 3.1). The designer then specifies the query using query builder notation.

```
dmin, dmax = pvd.val('date'), pvd.val('date')
c = pvd.val('chamber')
q = pvd.query.select('name', count()).from('T')
    .where(and(between('date', dmin, dmax),  eq('chamber', c)))
  .groupby('name')
```

She now makes two interactions. `i1` binds values to `dmin,dmax`; interacting with the sliders will rapidly update their bindings and `q`'s result should update within 20ms; we call this the *continuous* latency bound. Motivated by prior work [50], the system is allowed to take up to 2sec to update when the user first switches to the slider from a different interaction (e.g., the radio buttons); we call this the *switch-on* latency bound. `i2` binds `c` and should take 200ms for both continuous and switch-on latencies.

```
i1 = q.iact([dmin, dmax], 20, 2000)
i2 = q.iact([c], 200)
```

Finally, the designer specifies the memory constraints and network properties, and calls JADE using the approximate solver (Section 4.3.2) to quickly return a feasible solution (say, Figure 2(e)). If JADE was previously run with the same inputs, it returns the cached solution. Finally, she initializes the solution to precompute data structures and runs the application on `localhost`.

```
pvd.memory({ client: '100', server: '1000'})
pvd.network({ latency: '20', throughput: '10'})
app = pvd.optimize({ solver: "approx" })
app.init()
app.run('localhost:8000')
```

If there is no feasible solution, JADE finds a plan that meets the latency bounds and minimizes memory usage. The designer can decide whether to allocate more resources or redesign the interface.

2.2 Problem Setup

JADE takes as input a cloud database instance, the interface's DIFFPLANS, interactions with their latency constraints, and resource constraints. It then generates a physical execution plan spanning a client-server architecture that meets these constraints.

Since JADE focuses on the data layer, it models the interface as a set of DIFFPLANS (one for each chart), interactions, and their latency constraints. A DIFFPLAN δ is a plan that contains

choices $\delta.C = \{c_1, \ldots, c_n\}$, implicitly representing a set of query plans. A DIFFPLAN therefore generalizes normal query plans with the addition of *choice* expressions and operators (collectively called *choices*, see section 3.1 and section 3.2) that dynamically change expression and operator subtrees at runtime in response to user interactions. DIFFPLANs are amenable to static analysis and optimization. Similarly to parameterized queries, binding concrete parameter values resolves to a concrete query plan, but DIFFPLANs generalize choices from mere parameter values to expressions and sub-plans. To summarize, DIFFPLANs execution first passes bindings top-down to resolve choices and then bottom-up to execute the resolved plan.

In addition to *choices*, JADE introduces three classes of physical operators. The cloud operator is a source that issues queries to the cloud DBMS. Identity operators manage data movement but do not change contents, and include network and caching operators. Finally, data structure operators are an extensible abstraction to support custom visualization-specific data structures. When combined with identity operators, they enable JADE plans to control pre-computation, caching, and placement decisions between the server and client. Section 3.2 describes these operators in detail.

An interaction $x = (\{(\delta_1, C_1), (\delta_2, C_2), ...\}, L_c, L_{sw})$ can affect multiple charts, i.e., DIFFPLANs, and is expected to meet continuous (L_c) and switch-on (L_{sw}) latency constraints. To simplify the notation, we will treat x as a set of its DIFFPLANs. Thus, for each $\delta_i \in x$, the interaction specifies a subset of choices $C_i \subseteq \delta_i.C$ that it will bind, and $\delta_i.l_c(x)$ and $\delta_i.l_{sw}(x)$ are the interaction's continuous and switch-on execution latency for δ_i. The interaction's estimated overall continuous latency is the maximum across its DIFFPLANs $l_c(x) = \max_{\delta \in x} \delta.l_c(x)$, and similarly for switch-on latency, indicating the most the user would have to wait for any of the charts(i.e., DIFFPLANs) to update.

EXAMPLE 4. *Continuing the previous example in Section 2.1, the DIFFPLAN for the bar chart is illustrated in Figure 2(b). In other words,*

$$\delta = \gamma_{\text{name,count()}}(\sigma_{\text{date} \in [\$a,\$b]}(\sigma_{\text{chamber}=\$c}(\text{votes}))$$

Here, $\delta.C = [\$a, \$b, \$c]$ *has three choices, and binding them resolves into an executable query e.g.,* $\gamma(\sigma_{\text{date} \in [9/10,10/10]}(\sigma_{\text{chamber}='\text{house}'}(\text{votes}))$. *The range slider interaction* x_{range} *consists of a pair* $(\delta, \{\$a, \$b\})$ *with latency constraints* $L_c = 20$ *and* $L_{sw} = 2000$*; the radio button* x_{radio} *has a pair* $(\delta, \{\$c\})$ *with* $L_c = L_{sw} = 200$.

Since the execution latencies are not known during optimization, they must be estimated. This raises the natural question of whether JADE should optimize for the expected or upper-bound latency estimates. To this end, JADE uses both: it uses the upper-bound estimates to meet the designer's latency bounds and minimizes the expected latency, denoted with superscripts $l_\square^{\text{upper}}$ and $l_\square^{\text{avg}}$ respectively, where $\square \in \{c, sw\}$. Finally, $M_c(\delta)$ and $M_s(\delta)$ denote the client and server memory needed for the plan.

Finally, an interface $I = (\Delta, X)$ is a set of DIFFPLANs Δ and interactions X. The total client memory, $M_c(I)$, is the sum of $M_c(\delta)$ for all $\delta \in \Delta$, excluding the possibly shared cache among different δs (similarly for the server). We use $\widetilde{l_x}, \widetilde{M_c}, \widetilde{M_s}$ to denote estimates, and drop the argument (δ) if it is clear from the context.

2.3 Problem Definition

We now formally state the main PVD problem:

PROBLEM 1 (PHYSICAL VISUALIZATION DESIGN). *Given an interface* $I = (\Delta, X)$*, client/server memory constraints* $\mathbb{M}_c, \mathbb{M}_s$*, and network latency* l_{net} *and throughput* t_{net}*, return optimal physical plans* $\Delta^\star$ *to execute each interaction's corresponding query such that:*

- $l_c^{\text{upper}}(x) \le x.L_c \wedge l_{sw}^{\text{upper}}(x) \le x.L_{sw}\ \forall x \in X$

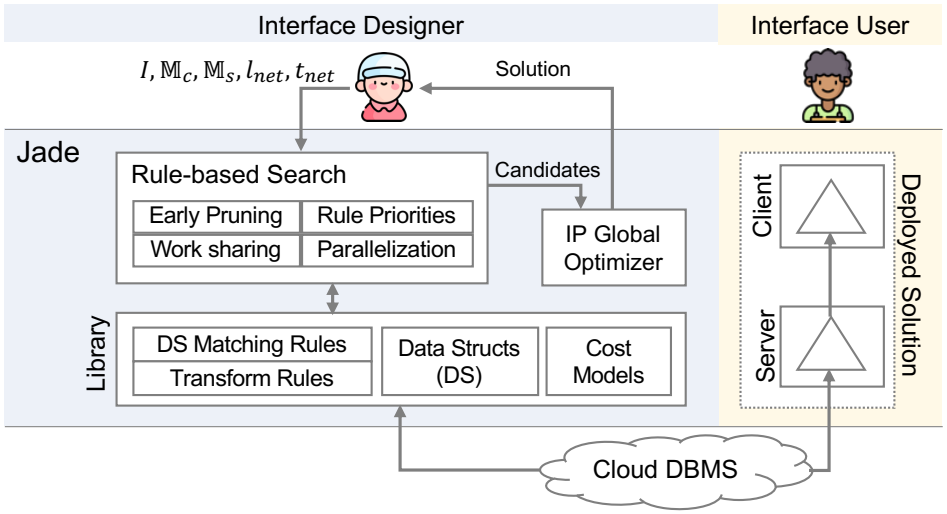

Fig. 3. Jade's two phases of usage.

- $M_c(\Delta^\star) \leq \mathbb{M}_c \wedge M_s(\Delta^\star) \leq \mathbb{M}_s$
- $\max_{x \in X} l_c^{avg}(x)$ *is minimized.*

When Jade cannot find a feasible solution, we relax the problem to find a solution that meets the latency guarantees and minimizes the weighed average of the relative client and server memory usage:

Problem 2 (Minimal Resource Search). *Given* $I = (\Delta, X)$, $\mathbb{M}_c$, $\mathbb{M}_s$, l_{net}, *and* t_{net}, *as well as a preference* $\alpha \in [0, 1]$, *return physical plans* $\Delta^\star$ *such that:*

- $l_c^{upper}(x) \leq x.L_c \wedge l_{sw}^{upper}(x) \leq x.L_{sw}\ \forall x \in X$
- $\alpha \times \frac{M_c(\Delta^\star)}{\mathbb{M}_c} + (1-\alpha) \times \frac{M_s(\Delta^\star)}{\mathbb{M}_s}$ *is minimized.*

2.4 System Overview

Figure 3 illustrates the two phases of Jade usage. The interface optimization phase (blue) translates the user's interface $I = (\Delta, X)$, constraints, and network properties into a feasible solution; the interface usage phase (yellow) deploys the solution by materializing data structures offline, instantiating a server that runs the portions of the physical plans below the network operator, and generating client-side JavaScript code that executes the portions of physical plans above the network operator.

The interface optimization phase uses rule-based search to find candidate physical plans that individually satisfy the constraints for each interaction and its affected Diffplan pair - (x, δ), and we developed a suite of heuristic optimizations to accelerate search. Each physical plan includes data structures, materialization policies based on dynamic or static caching, and network communication. Using these candidates, the global optimizer formulates an integer programming (IP) problem to find a combination of physical plans that 1) meet each interaction's latency constraints, 2) maximize data-structure sharing opportunities between plans, and 3) keep within global resource constraints.

The *Library* stores optimizer rules, an extensible set of data structures, and trained cost models. The rules include relational transformations (e.g., predicate push-down), choice-specific transformations, and rules to match subplans to data structures. Each operator has a cost estimation function which takes in standard database statistics and input cardinality estimates, and estimates the upper bound or average latency and memory usage.

The interface usage phase first computes or loads pre-computed static data structures into client/server memory. Jade then allocates a server and initializes the physical plans across the server and client. Each interaction sends bindings—a dictionary that maps choices to their corresponding values—to the appropriate physical operators. These operators execute the rest of the plan and pass the results to the interface renderer.

3 Library

A Diffplan generalizes normal query plans with the addition of *choice* expressions and operators. Data structures are modeled as operators. Jade has extensible libraries on data structure operators, rules, and cost estimation.

3.1 Choices

We now introduce the choice expressions and operators that Jade supports. All choices are nodes in a Diffplan, and serve to select a subset of its children or choose a literal from a domain. One perspective is that choices are a compact representation of a (possibly infinite) set of possible queries; binding the choices resolves the Diffplan to an element in the set, and interactions that change these bindings explore this space of possible queries.

3.1.1 Choice Expressions. Jade supports four choice expression nodes: `ANY` chooses one of its children, `OPT` toggles the presence of its child, `VAL` selects a value from a pre-defined domain, and `SUBSET` selects a subset of its children. A binding b is a dictionary that maps a node id to the bound value; the syntax `N(id;..)[b]` binds node N with b. Once N resolves itself, it recursively passes b to its child expression(s) using the following rules. Below, id refers to the node's unique id.

- `ANY(id; c:Expr+)` has n children, and resolves binding b[id] $\in [0, n)$ to its b[id]th child. For instance, `(a=ANY(0;1,3,5))[{0:1}]` resolves to `a=3`.
- `OPT(id; c:Expr+, default)` toggles its child. It resolves b[id] $\in (0, 1)$ to c if b[id] and `default` otherwise.
- `VAL(id; d:Domain)` checks that b[id] $\in$ d is within the domain d, and if so, resolves to b[id]. Jade supports two common domain types: an attribute A's active domain (e.g., $\texttt{VAL}_{\texttt{A}}(\texttt{id};\texttt{A})$) and a numeric range $\texttt{VAL}_{\texttt{R}}(\texttt{id};[\texttt{start},\texttt{end},\texttt{step?}])$ with optional step.
- `SUBSET(id; d:Domain, p)` is similar to `VAL` but resolves its binding b[id] $\subseteq$ d to a subset of its domain that will be used to construct an expression tree rooted at its parent p[1]. Our implementation supports three domain types: $\texttt{SUBSET}_{\texttt{A}}(\texttt{id};\texttt{A})$ uses an attribute's active domain; $\texttt{SUBSET}_{\texttt{R}}(\texttt{id};[\texttt{min},\texttt{max},\texttt{step?}])$ uses a numeric range, defined by a minimum, maximum, and optional step; $(\texttt{SUBSET}_{\texttt{E}}(\texttt{id};\texttt{Expr+})$ uses a set of expressions. `SUBSET` is useful to choose grouping expressions, projection expressions, lists, and filter clauses. For example, `SUBSET(id; a=1, b=2, AND)` can express `a=1` or `b=1` or `a=1 AND b=1`.

3.1.2 The `AnyOp` *Choice Operator.* The `AnyOp(Op+)` choice operator is analogous to the `ANY` expression, but it and its children are operators. This allows it to dynamically change query substructures as long as they are pre-defined—for instance, to choose from a set of source tables. Nesting `AnyOP` defines a combinatorial set of plans. `AnyOp` is considered both a logical and physical operator.

3.2 Physical Operators

Jade introduces a suite of physical operators for constructing execution plans. Unlike typical operators, a Jade operator takes a table or a binary blob typed to a data structure (Blob_{ds}) and an

[1]Jade implements `AND` and `OR` as parents because they are sufficient for the experiments. Any commutative and associative expression can be a parent.

expression list as input. It is bound to bindings b (non-choice operators may ignore the bindings) and returns a table or a typed blob.

$$\texttt{Op(input:Table|Blob}_{\texttt{ds}}\texttt{, es:Expr}^{*}\texttt{)[b:Bindings]} \rightarrow \texttt{Table|Blob}_{\texttt{ds}}$$

Physical operators are executable on the server, client, or both. In addition to `AnyOp`, Jade introduces two classes of physical operators: *Identity* operators and *Data Structure* operators.

3.2.1 Identity Operators. Identity operators do not change data contents; instead, they are used to manage data movement. Network, cloud, and cache operators are used to express materialized views and control data placement and caching.

The network operators `Send(input)` and `Receive(input)` follow the exchange operator [34] design to manage network communication and allow for split execution across the client and server.

The cloud operator is a unary operator with a logical Diffplan as its child. For instance, `Cloud(ANY(T1, T2))`'s child chooses between two tables. Whenever its child is resolved to a concrete plan through interaction bindings, the operator generates the corresponding query string and sends it to the cloud DBMS. Results are forwarded to its parent operator. The main restriction is that the child Diffplan cannot contain identity or data structure operators.

The cache operators are in-memory hash tables: the keys are bindings of its descendant choices, and the value is the result of its child operator given that binding. If there is a cache hit, the cached value is returned and the rest of the subplan is logically truncated. Otherwise, it passes the bindings to its child operator and adds its output to the cache following the replacement policy.

We implemented two popular types of caches. `SCache` is a static cache that allocates sufficient memory for all possible bindings of all choices in its subplan and pre-populates the cache offline. It essentially truncates its subplan. `DCache` is a dynamic cache that allocates memory for the most recent input table/blob and does no pre-population. When the input table/blob changes, it replaces the cache. Future work can explore other caches and policies, such as disk caches and dynamic caches with multiple slots.

3.2.2 Data Structure Operators. Jade supports an extensible library of data structures that are relied upon to accelerate interactions and meet latency guarantees. For our experiments, we implemented seven data structures: HashTable, B+ tree, R-tree, data cube, 1D and 2D cumulative data tiles [50], and materialized views.

Each data structure implements a `match(Operator)`→`matchState` function used during query optimization to check if it expresses a subplan. It returns a binary blob `matchState` used to initialize the data structure if there is a match and null otherwise. It also implements physical operators to create, use, and communicate the data structure during interface usage:

- `build(Table, matchState)`→$\texttt{Blob}_{\texttt{DS}}$ uses an input table and match state to construct a binary blob that encodes the data structure.
- `eval(`$\texttt{Blob}_{\texttt{DS}}$`, matchState)[b:Bindings]`→`Table` uses the blob from `build()`, matchState, and bindings to return a result table.
- `de/serialize(input:blob)`→`bytes` ensures that the data structure can be sent over the network. Otherwise, `build` and `eval` must be co-located.

Like other physical operators, each data structure also provides average and upper-bound cardinality, memory, and latency estimates.

Combining Identity and Data Structure Operators Identity operators can be interleaved with `build` and `eval` operator pairs to make fine-grained placement and caching decisions. For example, simply using `DCache` reproduces the caching strategy used by Falcon [50] to optimize the active widget, while `SCache` at the top of the plan full pre-computes all possible query results. Inserting a

network operator between `build` and `eval` offloads expensive data structure creation to a server. In short, combining identity and data structure operators creates a rich optimization space.

3.3 Optimizer Rules

Jade follows standard rule-based optimizer design [17, 22] to search the logical and physical plan space. Each rule consists of a pattern that matches a subplan, and a function to transform the match into a semantically equivalent subplan. Jade supports three classes of rules.

3.3.1 Adapted Rules. Existing transformation rules can be adapted to subplans with choices under straightforward semantics: a rule is valid if it is valid in all subplans encoded by the choices. In our implementation, we adapted three logical transformations: splitting conjunctive predicates, swapping the order of two filter operators, and swapping filter and group-by operators.

3.3.2 Data Structure Rules. Each data structure implements the `match(Operator)` method (Section 3.2) that the optimizer calls on each node in a candidate plan. If there is a match, the optimizer replaces it with a pair of `build` and `eval` operators with a logical `cache` operator sandwiched between them. The optimizer later chooses the physical cache operator (`SCache` or `DCache`).

3.3.3 Choice Rules. We designed specialized rules to push choices up or down the plan, specifically the `AnyOp` operator and `Any` expression[2]). We will describe the push up rules as other variations, and push down rules are straightword extensions.

For the `AnyOp` operator, pushing it above its parent operator P involves adding its parent to each of its children:

$$\mathsf{P}(\mathsf{AnyOp}(\mathsf{c}_1, \ldots, \mathsf{c}_n)) \rightarrow \mathsf{AnyOp}(\mathsf{P}(\mathsf{c}_1) \cdots, \mathsf{P}(\mathsf{c}_n))$$

For the `Any` expression, rather than reorder it within an expression tree, we push it out of the expression into a choice operator. Here, the operator `P` has a child operator `c` and a choice expression with n subexpressions. When we push the `Any` up, it results in n parent operators, each containing one of the subexpressions.

$$\mathsf{P}(\mathsf{c}, \mathsf{Any}(\mathsf{e}_1, \ldots, \mathsf{e}_n)) \rightarrow \mathsf{AnyOp}(\mathsf{P}(\mathsf{c}; \mathsf{e}_1), \ldots, \mathsf{P}(\mathsf{c}; \mathsf{e}_n))$$

Note that the choice expression can be anywhere in the expression tree and not just its root. While more sophisticated rules are possible, such as splitting a domain, we find these simple rules to be effective and leave a detailed analysis of the rule space to future work.

3.4 Cost Models

The latency and memory estimation for interactions differ from the traditional estimation in three ways. First, costs are interaction-centric rather than plan-centric because the same Diffplan can be bound by multiple interactions, so the same plan can have different latency estimates. Second, we distinguish between the first time the user manipulates an interaction (switch-on latency) and the latency during continuous interaction (continuous latency). Third, we strive to meet guarantees, so estimating expected latency and memory usage is insufficient: we need to estimate upper bounds.

In this subsection, we describe our approach to estimate the upper-bound continuous $\widetilde{l_c^{upper}}(x)$ and switch-on $\widetilde{l_{sw}^{upper}}(x)$ latency for an interaction x, as well as the expected latencies $\widetilde{l_c^{avg}}(x)$ and $\widetilde{l_{sw}^{avg}}(x)$. We start with per-operator estimation and then extend to a plan. The principles for estimating memory usage are the same.

[2]All choice expressions reduce to `Any`, so we focus on it.

3.4.1 Operator Estimation. Given an operator o, we estimate its execution time $\widetilde{o.t}$ by fitting a simple linear regression model whose features are the number of input and output columns, cardinality, size in bytes, and column types. We generate training data by executing 1,000 randomly generated execution plans for the six applications of all data size in section 5.1 covering all these operators for 20 times and collecting the ground-truth execution time along with the corresponding features. When the optimizer uses the model to estimate an operator's latency, we control how conservative or aggressive the cardinality estimates are to estimate the upper-bound or expected latency. Concretely, we control the conservativeness of the cardinality estimates by setting different types of predicate selectivity referring to the histogram statistics. For example, for upper bound estimates, we use the selectivity of the most common value. If there are multiple-attribute selections in conjunction, we use the smallest selectivity among them. For the average estimates, we use the selectivity of $\frac{1}{|\text{unique values}|}$. These result in $\widetilde{o.t^{\text{upper}}}$ and $\widetilde{o.t^{\text{avg}}}$ for the operator.

3.4.2 Plan Estimation. We first describe how to estimate the upper-bound latency of a plan, and then describe the minor change to estimate the expected latency. Briefly, we sum the upper-bound operator latencies $\widetilde{o.t^{\text{upper}}}$ along each path from root to leaf operator, and take the maximum. The only exception is that if the operator is a cache and there is a known cache hit, then it is treated as a leaf. Cache hits occur if the operator is an `SCache` or if we estimate the continuous interaction latency for a `DCache` operator. Otherwise, the cache operator is a no-op. To estimate the expected latency, we average over every path rather than take the maximum.

Example 5. *Figure 4 shows another* Diffplan δ *for the interface in addition to the three* Diffplans *in Figure 2(c-e).* δ *can calculate the interaction results for the slider* ① *and the radio* ②*. It builds a* `SCache` *to store the materialized view of the table T on the client, calculate the filter over chamber, and build a dynamic cache of a cube.*

$\delta.\widetilde{l_{sw}^{\text{upper}}}$(①) *will assume the* `DCache` *is not populated and thus equals to the summation of all operators' latency till the* `SCache` - 16ms = 1 + 10 + 5ms. $\delta.\widetilde{l_{c}^{\text{upper}}}$(①) *equals to 1 ms which will only need to run the* `Cube.eval`*. Here,* $\delta.\widetilde{l_{sw}^{\text{upper}}}$(②) *and* $\delta.\widetilde{l_{c}^{\text{upper}}}$(②) *are both* 16ms *since the plan executes from* σ *to the root.*

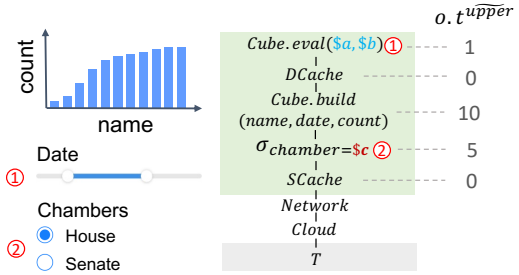

Fig. 4. Interface and its Diffplan: rightmost column shows the upper bound execution time of each operator.

4 Jade Engine Design

Recall that the input to Problem 1 is an interface I = (Δ, X), client and server memory constraints $\mathbb{M}_c$, $\mathbb{M}_s$, network latency l_{net} and throughput t_{net}, and the library of transformation rules $\mathbb{R}$. Δ is the set of logical Diffplans and each interaction x = ({(δ_i, C_i), . . . }, L_C, L_{SW}) ∈ X specifies the choices in a set of Diffplans that the interaction will bind, along with the latency constraints. The structure of our solutions is as follows: for every pair of interaction and logical Diffplan that it binds (x, δ), choose an optimal physical Diffplan that meets the interaction's latency constraints, such that

the total resources across all chosen physical DIFFPLANs meet constraints. We will first describe an exhaustive baseline approach, then describe pragmatic optimizations to run the optimizer interactively.

4.1 Exhaustive Baseline

The baseline runs two very slow steps—Candidate Search and Optimal Selection—that we will optimize in the subsequent subsections. Candidate search (Algorithm 1) loops through each interaction $x \in X$ and each diffplan δ_i that the interaction will bind, and finds all viable physical plans $\Delta_{x,i}$ using the following steps. It exhaustively applies the rules in $\mathbb{R}$, and for each plan, tries all assignments of static or dynamic cache between `eval` and `build` operators as well as other valid cache placements, enumerates all valid placements of `cloud` and `network` operators, and only keeps viable plans that satisfy the latency constraints - (L_c,L_{sw}).

Algorithm 1 Baseline Candidate Search

```
for x ∈ X do
    for δ_i ∈ x do
        Δ_{x,i} = {}
        for δ_phys ∈ exhaustive enumeration do
            Insert δ_phys into Δ_{x,i} if δ_phys is viable.
```

Next, optimal selection finds the best $\delta^{\star}_{x,i} \in \Delta_{x,i}$ for each pair of interaction and DIFFPLAN, so that the combination satisfies the memory constraints and minimizes the average latency estimates. It simply enumerates every combination.

4.2 Candidate Search Optimizations

Candidate search trivially parallelizes over every interaction, DIFFPLAN pair (x, δ); We also developed two effective pruning heuristics.

4.2.1 Work Sharing. Exhaustive enumeration for every (x, δ) pair can be wasteful because for a given DIFFPLAN δ, almost all search steps are independent of the interaction x. The only dependent decision is that the dynamic cache should never be above the choices that x will bind, because the cache would never be reused.

Thus, we run candidate search for each δ to enumerate all reachable physical plans; these plans insert a cache placeholder between each data structure `Build` and `Eval` pair and also enumerate other viable places. For each interaction x that binds δ, we try all static and dynamic cache assignments, and enforce that the `DCache` is above the choices that x binds. We add every viable plan as a candidate for (x, δ). If each DIFFPLAN is bound by N interactions, this reduces search cost by N×.

4.2.2 Push-ups First. Traditional query optimizers[22] are often staged so rules that almost always improve the query (e.g., predicate push-down) are applied before more general rules. Similarly, we first heuristically apply push-up rules so that all `AnyOp` and `Any` choices whose domains are not literals are at the top of the resulting plans. We do this because data structures typically do not try to match `AnyOp` and `Any` choices that are not literals, so their presence will cause false negatives, and moving them to the top makes matching data structures more likely. In addition, it is possible to search for candidates of an `AnyOp`'s child subplans in parallel, so pushing them up to the top maximizes parallelization opportunities.

We refer to each subplan under the `AnyOps` as a *basic plan* β_j because they only contain parameterized literals, which are familiar to traditional matching patterns. For each basic plan β_j, we

enumerate its set of viable physical plans $\Delta_{x,i,j}$ by exhaustively applying the rules and parallelize search across the basic plans. Joins are always pushed to the bottom and materialized using a cache operator to avoid computing joins during user interactions. We avoid join optimization, so exhaustive enumeration is manageable.

Example 6. *Below shows an example where the filter predicate contains* $\texttt{ANY}(\$1; a, b)$ *and* $\texttt{VAL}_R(\$2; [1, 5])$. *We push the* ANY *operation out of the expression tree into the* AnyOp *operator at the top. The transformation results in two basic plans—one for each* σ *plan:*

$$\sigma_{\texttt{ANY}(\$1;a,b)=\texttt{VAL}_R(\$2;[1,5])} \longrightarrow \texttt{AnyOp}(\$1; \sigma_{a=\texttt{VAL}_R(\$2;[1,5])}, \sigma_{b=\texttt{VAL}_R(\$2;[1,5])})$$

We preserve the ids of the choice nodes and do not flatten them so that the original bindings still apply. For instance, the binding $\{\$1:0, \$2:2\}$ *chooses* $\sigma_{a=2}$ *in both the original and transformed* DiffPlans. *Note that if the two choices had different latency constraints (e.g., $1 was 20ms and $2 was 200ms), then the latency of all plans is now* 20ms.

4.2.3 Additional Pruning Heuristics. First, we restrict each DiffPlan to at most one static cache; once it is placed, we do not consider any static or dynamic caches under it. Second, if a given static cache for a data structure exceeds the resource constraints $\mathbb{M}_c$ or $\mathbb{M}_s$, we prune any plan that contains that static cache.

Example 7. *Figure 5 shows an example of candidate searching for basic plans. In this example, we search for physical candidates* $\Delta_{\$2,i,2}$ *of the basic plan* β_2 *for interaction* $2. *By applying rules, here are two candidates among all the candidates the searcher finds: the candidate ① statically caches the raw table and calculates the filter operation on the server, while the candidate ② builds a hash table for attribute* b *on the server and statically caches it on the client for future queries. Although the second candidate requires slightly more memory due to the hash table, it offers significantly lower latency.*

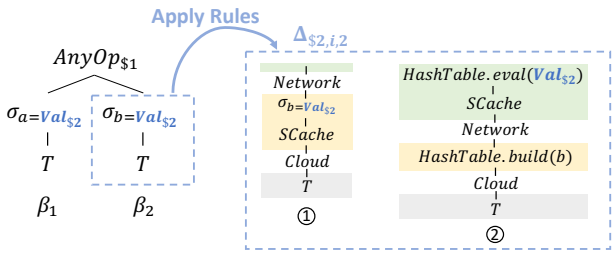

Fig. 5. Two candidates ① and ② for the basic plan β_2.

4.3 Optimal Selection

The output of candidate selection is a set of viable plans $\Delta_{x,i,j}$ for every interaction x and every basic plan β_j in each DiffPlan $\delta_i \in x$ after push-ups. Since all viable plans satisfy the interaction latency constraints, following Problem 1, our goal is to choose an optimal $\delta^{\star}_{x,i,j} \in \Delta_{x,i,j}$ such that the maximal $l_c^{avg}(x)$ across all interactions is minimized. We structure the problem around two constraints: ① for every interaction and basic plan pair, one plan is selected, and ② the total resource consumption over all plans is within server and client memory constraints $\mathbb{M}_c$, and $\mathbb{M}_s$.

Unfortunately, this problem is NP-hard because the optimal plan selection for different interactions and basic plans are not independent and may share data structures. This can be shown via a polynomial-time reduction from the NP-complete hitting set problem to optimal selection; we omit the proof due to space constraint.

We solve the optimal selection problem using integer programming, by translating ① and ② above into two sets of constraints. We model all dynamic and static caches in the candidate

plans using binary variables $\mathbb{D} = \{D_1 ... D_d\}$ and $\mathbb{C} = \{C_1 ... C_k\}$, respectively. Setting a variable to 1 chooses the corresponding static/dynamic cache. C_i.m and D_i.m denote the cache's memory usage (Section 3.4). We define a variable `avglatency` to denote the maximal $l_c^{avg}(x)$ across all interactions. The objective function is: minimize (`avglatency`).

To instantiate δ, each candidate set $\Delta_{x,i,j}$ in ① must choose 1+ candidates $\delta \in \Delta_{x,i,j}$ whose caches are built. This is expressed by the two Π terms, where C_δ and D_δ denote the set of static and dynamic caches in δ. The indicator function $\mathbb{I}()$ ensures that the estimated interaction latency $\widetilde{l^{avg}(x_j)}$ is smaller than `avglatency`.

$$\sum_{\delta \in \Delta_{x,i,j}} \{ \prod_{c \in C_\delta} c \times \prod_{d \in D_\delta} d \times \mathbb{I}(\delta.\widetilde{l^{avg}(x)} \leq \texttt{avglatency}) \} \geq 1 \quad \forall \Delta_{x,i,j} \qquad ①$$

Constraint ② ensures that the total memory across all selected plans is within client and server memory constraints ($\mathbb{M}_C$, $\mathbb{M}_S$). An `SCache` C_i pins its data structures, so its use is a constant C.m. However, an interaction x may materialize multiple `DCache` instances simultaneously and then be evicted when the user uses a different interaction. Thus, we only allocate the maximum `DCache` required for an interaction. We model server-side memory of static M_S^{SCache} and dynamic cache M_S^{DCache} as below, where $\mathbb{C}_S$ ($\mathbb{D}_{x,S}$) is the set of static (dynamic caches used by x) on the server.

$$M_S^{SCache} = \sum_{C_i \in \mathbb{C}^S} C_i \times C_i.m \qquad M_S^{DCache} = \max_x \sum_{D_i \in \mathbb{D}_{x,S}} D_i \times D_i.m \qquad ②$$

The total server memory consumption $M_S = M_S^{SCache} + M_S^{DCache}$ and its constraint is $M_S \leq \mathbb{M}_S$. The similar holds for the client memory $M_C = M_C^{SCache} + M_C^{DCache} \leq \mathbb{M}_C$.

4.3.1 Implementation Details. Jade uses Z3 [28], but the indicator function $\mathbb{I}()$ in ① causes Z3 to time out. To address this, we introduce an average latency threshold T_{avg} to remove all candidates in $\Delta_{x,i,j}$ whose average latency is larger than T_{avg}; this lets us remove $\mathbb{I}()$ and quickly find a solution. We use an outer loop that performs a binary search over the T_{avg} values between 0 and the largest latency constraint to find the smallest T_{avg} that produces a solution. The final value of T_{avg} is equivalent to minimizing `avglatency`.

The above formulation solves Problem 1 in Section 2.3 If no solution is found, we relax the hard memory constraint and instead optimize to minimize a weighted average of server and client memory usage, as described in problem 2. This approach helps the designer understand how much additional memory is necessary to meet the latency requirements.

4.3.2 Approximation. We also introduce an approximate variant by terminating the search after S seconds and choosing the top-K candidates with the smallest memory usage for each basic plan, where S and K are hyperparameters that control the trade-off between the quality and size of the candidate sets passed into the IP program. A larger S potentially finds better plans or more sharable subplans, while a larger K increases the accuracy of the results at the cost of slower integer programming. We evaluate this approach in section 5.6 and find that it greatly accelerates optimization without discernible impact on the final solution or application.

5 Evaluation

We compare Jade to visualization frameworks and PDD tools in how well they optimize a variety of data interfaces, latency bounds, and resources. Jade is more expressive than visualization frameworks because it can mix-and-match optimizations, and interactively finds feasible plans over a wider range of constraints than PDD.

5.1 Setup

Setup. We implemented our execution engine by connecting off-the-shelf components, such as the Acero operator for traditional query operator execution [11], open-source data structures like R-trees [52], and some manually implemented data structures. We used a client-server setup where each machine runs Ubuntu 22.04, with 2.4GHz Intel CPUs and 128GB of memory (we restrict their memory in the experiments). The server communicates with Azure SQL Database [48] [3] that stores the base data. The network latency is 0ms and throughput is 10MB/s.

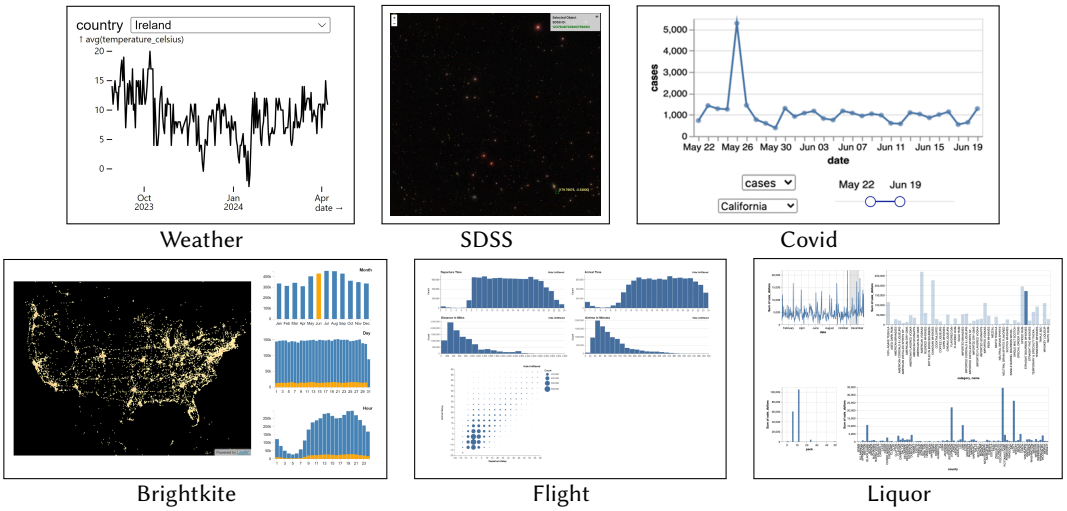

Fig. 6. Screenshots of the six representative data interfaces.

Interface	Complexity	V	I	I/V	V/I	C/δ	Description
Weather	Easy	1	1	1	1	1	Temperature changes by country.
SDSS	Easy	1	1	1	1	4	Zoomable scatterplot of stars.
Covid	Medium	1	3	3	1	3	Covid case/death trends of different counties.
Brightkite	Medium	4	4	1	3	7	Geographical distribution of Brightkite user check-ins changes by month, day, and hour, with month, day, and hour distribution patterns varying across different locations and time periods.
Flight	Hard	5	5	4	4	8	Binned aggregate counts are computed for departure time, arrival time, distance, and airtime. A heatmap visualizes the counts across arrival and departure times. Each chart features a linked brush that enables cross-filtering across all other charts.
Liquor	Hard	4	4	3	3	4	Study the liquor sales trends along packs, county, categories and date.

Table 1. Visualization applications ported to Jade. Attributes include numbers of views (V), interactions (I) interactions per view (I/V), views per interaction (V/I), choices per Diffplan (C/δ).

Baselines. Mosaic [38] is a state-of-the-art Javascript visualization framework that leverages data cubes and DuckDB [42] to accelerate interactive visualizations. We run the DuckDB on the server side. Azure SQL Database [47] (Azure) implements a state-of-the-art physical database design system. Given a query workload, it requires between 30 minutes and 72 hours [47] to propose optimized data structures for query execution. Prior to latency evaluation, we employed a simulator that mimics human interactions with interfaces to generate a query workload representative of each data interface. These queries were submitted to Azure SQL Database, allowing the automatic tuning feature sufficient time to construct its preferred data structures. Finally, interface queries

[3] We do not enable the automatic tuning when using Azure SQL Database as Jade's cloud database.

were executed against the database to formally benchmark the interaction latency. Neither system supports latency constraints, so we run their optimizations and report the resulting query latencies.

Data Interfaces. Jade is the first system to express and automatically optimize a wide range of data interfaces, and so we lack a preexisting workload to evaluate. Thus, we sought out a set of representative data interfaces that varied in domain and interface complexity. This survey resulted in six representatives: weather analysis from Kaggle [56], Sloan Digital Sky Survey [58], Google's Covid visualization [33], interactive cross-filtering interfaces to analyze Brightkite check-ins [35] and Flight delays [50] from academia, and a cross-filter visualization that analyzes Iowa liquor sales on Observable [40]. These representatives also subsume the multitude of interfaces created by frameworks such as Streamlit [8] and Hex [7].

While the data for all interfaces was available, we manually translated each interface into Diffplans. We define complexity based on if (1) each interaction affects one view, and (2) each view is affected by one interaction. An interaction that affects >1 views must schedule and concurrently execute queries. A view affected by >1 interactions means a Diffplan contains multiple choices and may need dynamic caching to avoid a combinatorially large static cache. We say an easy interface satisfies (1,2), a medium interface satisfies either, and a hard interface satisfies neither (Table 1). Figure 6 shows the screenshots of these interfaces.

We scaled each interface's database to vary between 1-100M records (10M default) using Synthetic Data Vault [62]. Based on prior HCI and visualization work [13, 15, 31, 43, 44, 50, 51, 60], we chose three latency bounds: fast (20ms), medium (200ms), and slow (2s). Widgets such as brush and range slider interactions are set to fast because they rapidly generate many queries per second and are sensitive to latency fluctuations [43]; all other interactions were set to medium. We set the switch-on latency to slow for all interactions. This is denoted by the default latency setting L1.

5.2 Comparison with Baselines

We first compare how Jade, Mosaic, and Azure optimize each interface under varying latency and resource constraints. In addition to the default L1 latency setting, let L2 and L3 be where all continuous latency bounds are respectively fast and slow; switch-on bounds are always set to slow. RS1 (100MB client, 1GB server) and RS2 (1GB client, 10GB server) represent lower and higher memory resources.

Table 2 summarizes whether each system found a feasible solution. With low resources (RS1), Mosaic only runs the Flight interface (which it was designed for), while Azure only supports the Liquor interface, but only if all latency bounds are 2s (L3). With high resources (RS2), Azure did not improve, while Mosaic only supports 55% of the settings. In contrast, Jade found feasible plans for all interfaces—even Brightkite—under RS1. Under RS2, Jade supported all settings because it can mix and match optimizations, placement, and caching policies in a more flexible way than Mosaic.

5.3 Is Feasible Really Feasible?

Since Jade evaluates feasibility using cost models, we first empirically study whether feasible designs actually meet the desired constraints. We then study the trade-off between how conservative the cost model is and the likelihood Jade finds a feasible plan.

5.3.1 Do Feasible Plans Really Meet Constraints? We answer in the affirmative by reporting the resource and latency constraints compared with the actual memory usage and interaction latency[4]. We use L1 and all pairs of client ($M_c \in [100MB, 1GB]$) and server ($M_s \in [1GB, 10GB]$) memory.

[4]Actual latency is measured as the time between issuing the query on the client and receiving the results on the client.

Figure 7(top) reports, for each interface (y-axis) and interaction (points), the ratio of actual latency over the latency constraint (x-axis). We see that all ratios are below 1, meaning the produced plans meet the constraints. Figure 7(bottom) reports the percent of client and server resources actually (○ for client, ▲ for server), and we find that JADE consistently meets the resource bounds. In fact, the simpler interfaces (e.g., Weather, SDSS, Covid) take negligible resources. These results hold for L2 and L3 settings as well.

		Weather			SDSS			Brightkite			Covid			Flight			Liquor			Optimized
		L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3	%
PVD	**RS1**	✓	✓	✓	✓	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**88%**
	RS2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**100%**
Mosaic	**RS1**	-	-	-	-	-	-	-	-	-	-	-	-	✓	✓	✓	-	-	-	**16%**
	RS2	✓	-	✓	✓	✓	✓	-	-	✓	-	-	-	✓	✓	✓	-	-	✓	**55%**
Azure	**RS1**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	**5%**
	RS2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	**5%**

Table 2. Can JADE, Mosaic, and Azure find feasible solutions?

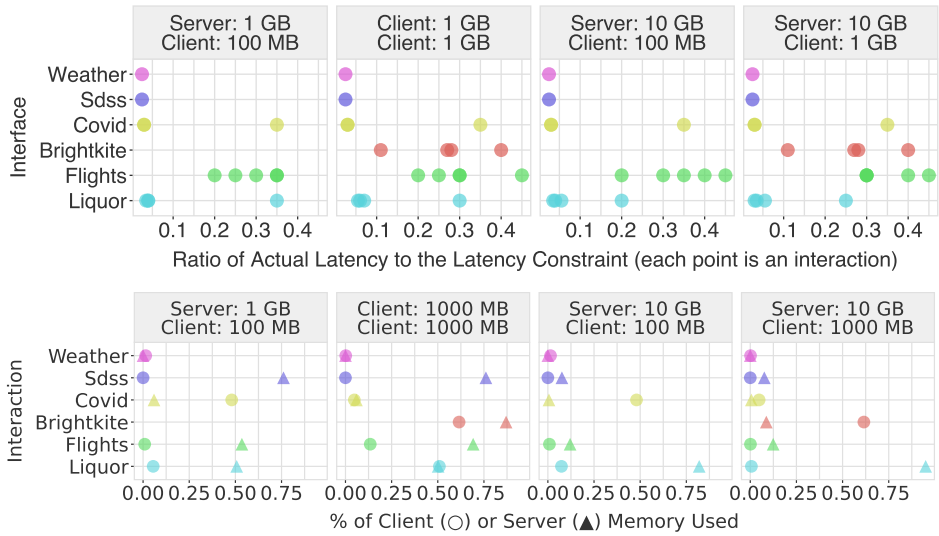

Fig. 7. The ratio of each interface's (top) actual latency to latency constraint, where each interaction is a separate point, and (bottom) percent of available memory used. The actual latency and memory use is empirically always lower than the constraint.

5.3.2 Cost Model and Feasibility. Our cost models are trained to predict latency or memory from input cardinality, and JADE by default uses the conservative `Upper`-bound cardinality estimate that uses attribute statistics (Section 3.4.1). However, an overly conservative estimate will likely lead to fewer feasible plans because the estimated latency or memory is more likely to exceed the constraints, while an aggressive estimate will have more feasible plans but the *actual* latency or memory usage is likely to exceed the constraints. We study this trade-off by using three additional estimators. `NoFilter` assumes all selectivity is 1 and ignores data statistics, `Avg` estimates the average cardinality by assuming uniform value distributions, while `Avg0.5` divides `Avg` by 2.

Figure 8 reports a microbenchmark that compares the estimated and actual latencies for each of the four cost models; the diagonal is where the estimate and actual are the same. After running JADE under L1 and all nine resource constraints, we recorded every operator's real and estimated execution times, and repeated this for 20 runs. We color code points by whether they are above or below the diagonal. All points to the left of 20ms are colored red because they are already

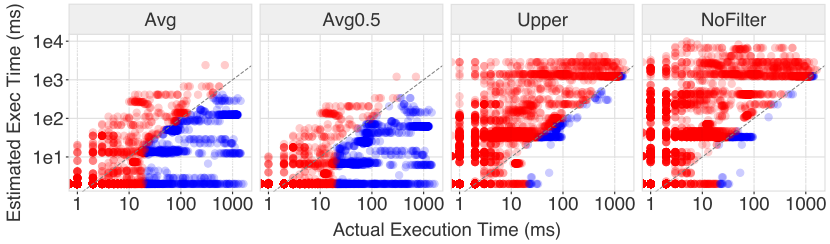

Fig. 8. Actual execution time vs. estimated execution time using four different cost models. Red points are either above the diagonal or correspond to actual execution times less than 20 ms. These indicate that the estimated execution time is greater than the actual execution time, or that the actual execution time is very fast, making it acceptable even if the estimate is smaller. The blue points represent all other cases.

fast irrespective of the estimate; the points of concern are those that are estimated to be fast but are actually slow. `NoFilter` and `Avg0.5` wildly over- and under-estimate the latency, respectively. `Upper` and `Avg` are both near the diagonal, however `Upper` tends to slightly over-estimate, which is preferable for meeting constraints.

Cost estimates also affect whether Jade finds a feasible plan at all and whether the feasible plan violates any constraints. To test this, we fixed latency constraints to L1, and generated all combinations of resource constraints where client$\in$ {0, 100MB, 1GB} and server$\in$ {500MB, 1GB, 4GB}. Note that Jade can either return (C1) no feasible plan, (C2) a feasible solution that meets constraints, or (C3) a feasible solution that violates some constraint. We count the number of each case to calculate measures for feasibility ($\frac{C2+C3}{C1+C2+C3}$) and accuracy ($\frac{C2}{C2+C3}$). Figure 9 shows that `Avg` (`Avg0.5`) find more feasible plans, but only <65% met the constraints. `NoFilter` always meets constraints but only finds solutions for <50% of problems. `Upper` balances accuracy and feasibility, so that 80% of problems are solvable, and nearly all feasible plans meet constraints. Thus, the rest of the experiments continue to use `Upper` as the default.

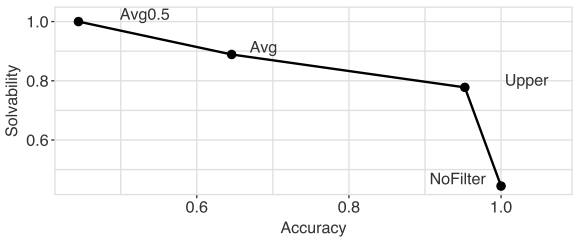

Fig. 9. Accuracy-solvability trade-off for four cost models.

5.4 Scalability to Large Databases

We scaled the database for each application from 1M to 100M records under the default `L1` and `RS2` constraints. Figure 10 reports the interaction latency for 9 representative interactions from across the interfaces to highlight the range of scalability trends. The dotted line is the interaction constraint, and we report Mosaic and Azure as baselines. Jade can always meet the latency bound irrespective of database size because it properly caches the necessary data structures. Azure's line reinforces the fact that *Cloud DBMSes are the slow path*. Mosaic exceeds the latency constraint in 5/6 interfaces for different reasons. For Weather and Liquor, it proposes no data structures, it cannot express the Covid interface (its line is not shown), and it proposes data structures for Brightkite and Flight that still require executing queries over increasingly large datasets.

5.5 The Pareto Curve for Resource vs Feasibility

Jade makes it possible to identify the Pareto curve of the minimum resources needed to meet latency constraints. This is akin to a resource profile for a designer's interface. Under L1 constraints,

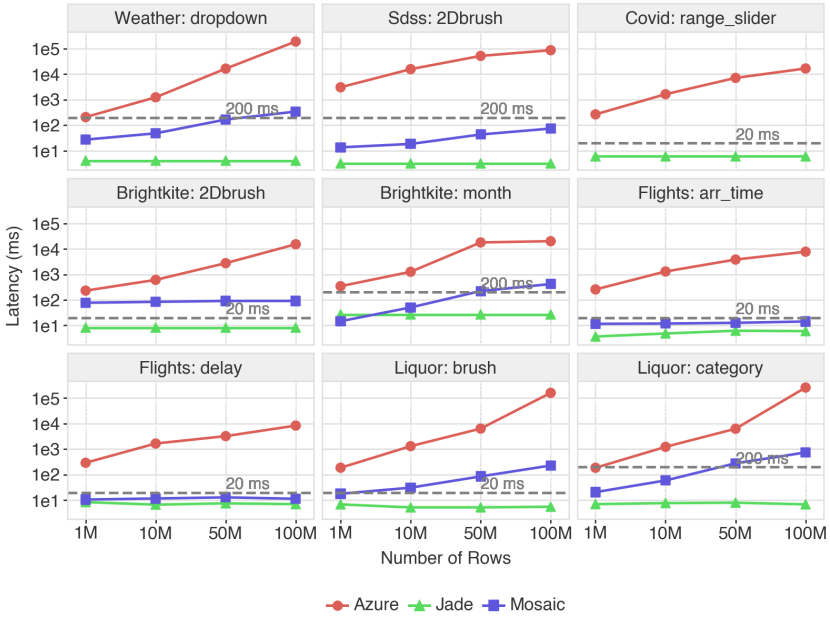

Fig. 10. Interaction latency as database size grows.

Fig. 11. The minimum resources needed to meet each interface's latency constraints.

we vary client memory from 0 to 2GB; for each client setting, we find the minimum server memory that finds a feasible solution.

Figure 11 shows three resource profiles. Interfaces that don't require resources exhibit a flat line (weather). Those that simply require a fixed-size data structure exhibit a ⌥ shape because the main decision is whether to cache on server or client (e.g., Liquor, SDSS, Covid, Flight). Brightkite exhibits a smooth shape because it balances different usable data structures and their placements. If the network is fast, the client can often have no memory as long as the server has sufficient resources; this is not the case for Brightkite because even with unbounded server memory, the network alone violates latency constraints.

5.6 Optimization Runtime

We now report the optimization time needed to find a feasible plan when using L1 and RS2 constraints. We compare the full optimization algorithm and the approximate solution described in Section 4.3.2, which stops search after S = 2 seconds [5] and uses the K=100 candidates with the smallest memory requirements for each DIFFPLAN as inputs to the IP. The feasible plans found by both approaches exhibit similar average interaction latencies (e.g. 2.3243 ms vs 2.3244 ms for full and approximate) and the actual memory usage stays within resource constraints.

[5]For liquor application, it appears to be longer than 2s because there needs some time to exit the search.

Figure 12 reports the search and IP runtimes for both methods. While most interfaces are solved well within 250ms, approximation speeds up optimization for the Liquor interface by around 3× (18.4s → 5.6s) because it places a hard time cap on the search procedure, and also bounds the size of the IP problem to be linear with the number of DIFFPLANS. The ability to return designs within seconds, and often in a few hundred milliseconds, can empower designers to incorporate JADE within the design loop.

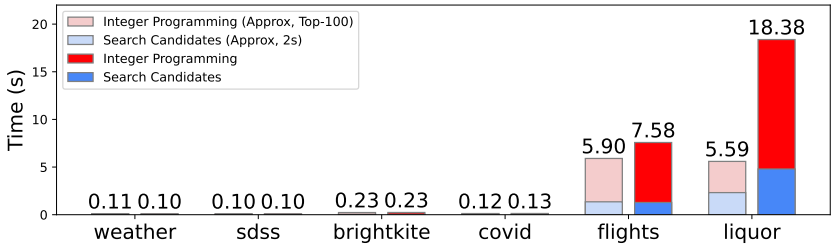

Fig. 12. JADE Runtime with or without approximation.

5.7 Case Study

This section will show case how PVD optimizes the NYTimes Covid interface [4]. We will use the "Hot Spots" part of the NYT interface described next as an example of how JADE can reproduce their optimizations and flexibly re-optimize the architecture in response to interface changes.

Figure 13 visualizes county-level Covid-related statistics aross the U.S. The top tab changes the statistics shown: 1) Hot spots (avg cases/100K people in the past week); 2) hospitalizations per 100K people); 3) Vaccination rates; 4) Cases per capita; and 5) Deaths per capita. Hovering over a county renders a tooltip with details. The toolip for "Hot spots", for instance, shows two statistics (# cases/day, # cases/100K people) and the past 14-days as a line chart.

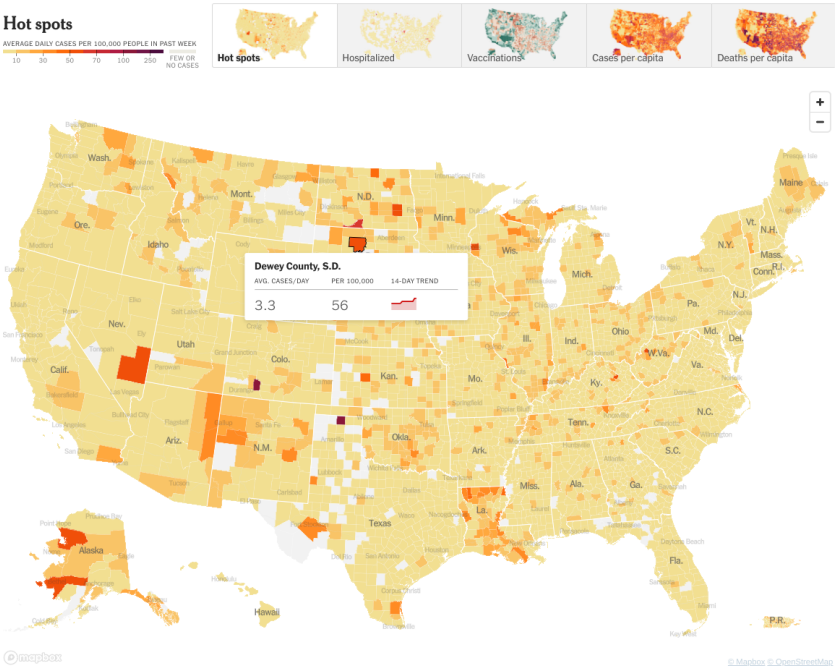

Fig. 13. The New York Times Covid interactive map shows five different distributions over U.S. controlled by the tab interaction. The tooltip interaction on each chart will show more specific information.

NYT Implementation. The above interactions are designed to respond within dozens of milliseconds. The hot spots chart is precomputed as an image, the two tooltip statistics are cached on the client, and the precomputed 14-day trends are cached on the server and fetched on hover.

Using the Jade Optimizer. The above system design can be easily reproduced via optimization; the other visualizations follow similarly. Hot spots consists of four queries to calculate each of the statistics described above, and they reference one VAL choice node that specifies the county:

```
c = pvd.val('county')
q1 = pvd.query.select('county', 'avg(cases/pol*100000)')
   .from('covid').where(gt('date', today() - '7 days'))
   .groupby('county')
q2 = pvd.query.select('avg(cases)').from('covid')
    .where(and(gt('date', today() - '7 days'), eq('county', c)))
q3 = ... // the #cases per 100K query is similar to q1 and omitted
q4 = pvd.query.select('date', 'cases').from('covid')
    .where(and(eq('county', c),  gt('date', today() - '14 days')))
```

The tooltip interaction should respond within 20ms (the tab interaction is omitted), and we specify the expected memory available and network properties.

```
i1 = iact(c, 20)
pvd.memory({client: 1, server: 1000})
pvd.network({latency: 10 , throughput: 1})
```

Figure 14 shows that Jade reproduces the manually optimized plans for each of the three queries above ($\delta_i^\star$ for qi). $\delta_1^\star$ precomputes the map visualization's data in the cloud DBMS and caches them on the client. To show the tooltip statistics for a selected county, $\delta_2^\star$ precomputes all counties' statistics, builds a hash table on county, and caches it on the client. $\delta_4^\star$ precomputes the 14-day trends, caches an index over them on the server. Thus, the 14-day trend data incurs a network roundtrip but no computation cost.

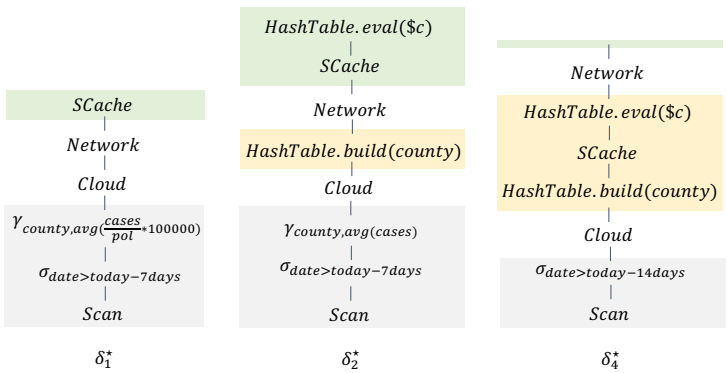

Fig. 14. Jade optimized plan $\delta_1^\star$, $\delta_2^\star$, $\delta_4^\star$ for q1, q2, q4 respectively. The grey part represents the computation on the database, the yellow is the server, and the green is the client side.

Jade Supports Iterative Design. The designer wants users to see historical statistics by choosing a desired date using a slider. Unfortunately, the previous optimizations are no longer applicable because they display fixed recent statistics instead of the historical data that users can specify using the slider. With Jade, the designer simply defines a new choice d to represent the selected date and include it in the affected queries (e.g., q1 shown below, and states its latency should be 200ms.

```
d = pvd.val('date')
```

```
q1 = pvd.query.select('county', 'avg(cases/pol*100000)')
  .where(between('date', sub(d, '7 days'), d))
  .groupby('county')
i2 = iact(d, 200)
```

Jade updates the system architecture in seconds. Now Jade outputs per-interaction optimal physical plans $\Delta^{\star}$. For the tooltip interaction `i1`, Jade proposes a B+ tree index on a composite key `(county, date)` cached on the server side. When the county changes, it will fetch the corresponding data for `q2-4` respectively and do the following computation of `q2-4` on the fly and return the results. The returning data is small - one integer for `q2, q3` and 14 integers for `q4`, so the network is fast and it satisfies the fast latency contraints. For the date slider, Jade proposes a cumulative data tile to accelerate the recent 7 days case computation for all the counties on the server. When the date changes, it will query the data tile and return the map over the network. The map is relatively large, so the interaction is not that fast, but is still within <200ms. When the date changes, `q2-4` still use the B+ tree to accelerate the query execution. As we can see, Jade quickly helps redesign the system architecture in seconds, whereas manual redesign and reimplementation could take days.

6 Related Work

Visualization Interface Query Representation Most interactive, SQL-based dashboarding tools, including Tableau, VizQL, Vega-lite, and Mosaic [36–38, 57, 59], can dynamically generate SQL that is executed in a DBMS. However, they do not specialize optimizations for each interface design, nor do they synthesize an end-to-end client-server execution plan to meet latency guarantees.

Prior works PI2 [24, 25] and DIG [23] introduced a compact representation for analysis tasks, respectively termed difftree and data interface grammar. This representation statically encapsulates query analysis while maintaining the correspondence between user interactions and queries using *Choice* nodes. However, these approaches are focused on syntactic strings and do not consider the execution query plan. This paper extends this concept by introducing Diffplan, which mimics a query plan and can be programmatically expressed using a query builder API common in data-frame systems like Spark, Pandas. Diffplan retains the choice-based abstraction between interactions and queries, but can be easily integrated into a standard query optimizer and executor framework.

Visualization-specific Optimizations Faster engines [27, 41, 53], data cubes [44], query rewriting and materialized views[63], precomputation / pre-aggregation[38], spatial indexes[60] all reduce query latencies for narrow classes of queries. Jade is designed to be extensible and, by specifying a matching pattern and cost model, can incorporate these techniques into the optimization framework.

The main exceptions to this are sampling and approximation techniques [9], which we do not consider because they change the semantics of the output visualization, and pre-fetching [14, 49], which is orthogonal to choosing data structures and execution plans. We leave incorporating these techniques into Jade to future work.

Parameterized Query Optimization or PQO [18, 39, 61] enumerates query templates to be compiled by databases, with desirable query plans cached across a range of potential parameterizations. Jade differs in two key ways; first, it allows safe parameterizations beyond simple literals extending into whole expressions and subqueries. Second, Jade encodes program-level context to guide data structure selection, while PQO is strictly concerned with query planning, not physical design.

Physical Database Design (PDD): PDD also uses access patterns in a query workload to choose indexes, partitions, and views to accelerate the queries [10, 21, 32, 45, 64]. Online database design (ODD) [19, 47**?**] monitors queries over time and balances reconfiguration and future query benefits. However, they are not applicable here because 1) they do not provide latency guarantees, 2) they operate internal to the cloud DBMS yet the DBMS is the slow path, particularly for low latency

interactions, and 3) ODD time scales are on the order of hours and the user experience suffers before optimization and during reconfiguration.

7 Limitations

We now describe current limitations and future directions.

Choice Independence. Jade assumes that choices are independent (e.g., choosing state doesn't affect the county dropdown), so the query space for a Diffplan is the cross-product of the domains of all choices. This forces Jade and the data structures to be more conservative than needed, and removing this independence assumption can expand the set of feasible plans that Jade can find.

Fast Rendering. Jade assumes that rendering time is negligible. While valid in data visualization, rendering can be expensive in domains like scientific visualizations [29].

Read-only Data. While Jade currently assumes read-only data, if the data structures were incrementally maintainable, then using Differential Dataflow [46] or DBSP [20] can maintain the Diffplans. Jade supports capacity planning where designers change statistics to check feasibility.

8 Conclusion

Physical Visualization Design (PVD) introduces design independence for interactive data visualization applications by decoupling the design, logic, and interactivity requirements of an interface from the system optimizations and execution architecture needed to meet interactivity and resource requirements. This paper formalized the PVD problem and described the first such system called Jade. Despite making stronger latency guarantees than physical database design, we surprisingly find that these guarantees eliminate a large class of difficult query optimization problems—namely join ordering—and thus make the problem tractable.

The core abstraction centers around the Diffplan—a compact representation of the queries that supply data for a view or chart in the interface. By extending logical query plans with *Choice* operators and expressions, they serve as targets for interactions to bind, but have well-defined domains and semantics that are amenable to analysis. Jade shows that rule-based optimization is effective at finding feasible physical execution plans that satisfy latency constraints, and uses integer programming to find a set of feasible plans that meet global resource budgets. Simple heuristics enable Jade to find feasible solutions within seconds. Further, Jade only needs to fit cost models for a given operator once, and can then use different cardinality estimators to control how aggressively or conservatively to estimate an operator's execution time.

Across a diverse range of six data interfaces, we demonstrate that as compared to state-of-the-art visualization systems like Mosaic and physical database design tools like Azure's SQL Database, only Jade can find feasible solutions under a wide range of latency and resource constraints. Feasible solutions nearly always meet latency and resource constraints in practice, and are often much faster and use fewer resources than estimated. For these reasons, we believe Jade can be an essential tool in a designer's toolbox.

Acknowledgments

This research received funding from multiple sources, including National Science Foundation grants (NSF #1845638, #1740305, #2008295, #2106197, #2103794, #2312991) as well as corporate support from Amazon, Google, Adobe, and CAIT. The views and conclusions presented here are those of the authors and should not be interpreted as representing the official positions of the funding organizations.

References

[1] 2014. Project Jupyter. https://jupyter.org. Accessed: 14 January 2025.
[2] 2020. Metabase. https://www.metabase.com. Accessed: 14 January 2025.
[3] 2020. Retool. https://retool.com. Accessed: 14 January 2025.
[4] 2021. Coronavirus in the U.S.: Latest Map and Case Count. https://www.nytimes.com/interactive/2021/us/covid-cases.html. Accessed: 14 January 2025.
[5] 2021. COVID-19 Dashboard by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University (JHU). https://coronavirus.jhu.edu/map.html.
[6] 2021. Tableau. https://www.tableau.com. Accessed: 14 January 2025.
[7] 2024. Bring everyone together with data. https://hex.tech.
[8] 2024. A faster way to build and share data apps. https://streamlit.io/.
[9] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In Proceedings of the 8th ACM European Conference on Computer Systems. 29–42.
[10] Ioannis Alagiannis, Debabrata Dash, Karl Schnaitter, Anastasia Ailamaki, and Neoklis Polyzotis. 2010. An automated, yet interactive and portable DB designer. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 1183–1186.
[11] Apache Arrow Developers. 2025. Streaming Execution in Apache Arrow. https://arrow.apache.org/docs/9.0/cpp/streaming_execution.html. Accessed: 2025-01-20.
[12] Michael Paul Armbrust. 2013. Scale-Independent Relational Query Processing. University of California, Berkeley.
[13] Dana H Ballard, Mary M Hayhoe, Polly K Pook, and Rajesh PN Rao. 1997. Deictic codes for the embodiment of cognition. Behavioral and brain sciences 20, 4 (1997), 723–742.
[14] 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. 1363–1375.
[15] 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. 1571–1587.
[16] 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 27, 2 (2020), 1128–1138.
[17] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. Proceedings of the 2018 International Conference on Management of Data (2018). https://api.semanticscholar.org/CorpusID:3554746
[18] Pedro Bizarro, Nicolas Bruno, and David J DeWitt. 2008. Progressive parametric query optimization. IEEE Transactions on Knowledge and Data Engineering 21, 4 (2008), 582–594.
[19] Nicolas Bruno and Surajit Chaudhuri. 2006. An online approach to physical design tuning. In 2007 IEEE 23rd International Conference on Data Engineering. IEEE, 826–835.
[20] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: Automatic incremental view maintenance for rich query languages. arXiv preprint arXiv:2203.16684 (2022).
[21] Wei Cao and Dennis Shasha. 2013. AppSleuth: a tool for database tuning at the application level. In Proceedings of the 16th International Conference on Extending Database Technology. 589–600.
[22] Meenu Chawla and Vinita Baniwal. 2018. Optimization in the catalyst optimizer of Spark SQL. Turkish Journal of Electrical Engineering & Computer Sciences 26, 5 (2018), 2489–2499.
[23] Yiru Chen, Jeffrey Tao, and Eugene Wu. 2023. DIG: The Data Interface Grammar. In Proceedings of the Workshop on Human-In-the-Loop Data Analytics. 1–7.
[24] Yiru Chen and Eugene Wu. 2020. Monte Carlo Tree Search for Generating Interactive Data Analysis Interfaces. ArXiv abs/2001.01902 (2020).
[25] Yiru Chen and Eugene Wu. 2022. PI2: End-to-end Interactive Visualization Interface Generation from Queries. In Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 1711–1725. doi:10.1145/3514221.3526166
[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. Introduction to algorithms. MIT press.
[27] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In Proceedings of the 2016 International Conference on Management of Data. 215–226.
[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 337–340.

[29] Thomas A Defanti and Maxine D Brown. 1991. Visualization in scientific computing. In Advances in Computers. Vol. 33. Elsevier, 247–307.
[30] Cube Dev. 2024. Cube: The Semantic Layer for Building Data Applications. https://cube.dev/ Accessed: 2024-07-22.
[31] 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. 1555–1569.
[32] Schkolnick Finkelstein, Mario Schkolnick, and Paolo Tiberio. 1988. Physical database design for relational databases. ACM Transactions on Database Systems (TODS) 13, 1 (1988), 91–128.
[33] Google. 2022. Google COVID Interface Search. https://www.google.com/search?q=google+covid+interface. Accessed: 2022-08-13.
[34] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In Proceedings of IEEE 9th International Conference on Data Engineering. IEEE, 209–218.
[35] Stanford Visualization Group. 2024. ImMens: Brightkite Dataset Demo. https://vis.stanford.edu/projects/immens/demo/brightkite/. Accessed: 2024-08-13.
[36] Pat Hanrahan. 2006. Vizql: a language for query, analysis and visualization. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data. 721–721.
[37] Jeffrey Heer and Michael Bostock. 2010. Declarative language design for interactive visualization. IEEE Transactions on Visualization and Computer Graphics 16, 6 (2010), 1149–1156.
[38] Jeffrey Heer and Dominik Moritz. 2024. Mosaic: An Architecture for Scalable & Interoperable Data Views. IEEE Transactions on Visualization and Computer Graphics 30, 1 (2024), 436–446. doi:10.1109/TVCG.2023.3327189
[39] Arvind Hulgeri and S Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In VLDB'02: Proceedings of the 28th International Conference on Very Large Databases. Elsevier, 167–178.
[40] Iowa. 2024. Iowa Liquor Dataset / Data Analysis. https://observablehq.com/d/3ce1ac2526c20515
[41] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In 2011 IEEE 27th International Conference on Data Engineering. IEEE, 195–206.
[42] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. 2022. DuckDB-wasm: fast analytical processing for the web. Proceedings of the VLDB Endowment 15, 12 (2022), 3574–3577.
[43] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. IEEE transactions on visualization and computer graphics 20, 12 (2014), 2122–2131.
[44] 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.
[45] Cristina Maier, Debabrata Dash, Ioannis Alagiannis, Anastasia Ailamaki, and Thomas Heinis. 2010. Parinda: an interactive physical designer for postgresql. In Proceedings of the 13th International Conference on Extending Database Technology. 701–704.
[46] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow.. In CIDR.
[47] Microsoft. 2024. Automatic tuning in Azure SQL Database. https://learn.microsoft.com/en-us/azure/azure-sql/database/automatic-tuning-overview?view=azuresql Accessed: 2024-07-22.
[48] Microsoft. 2024. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database. Accessed: 2024-09-02.
[49] Haneen Mohammed. 2020. Continuous prefetch for interactive data applications. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2841–2843.
[50] Dominik Moritz, Bill Howe, and Jeffrey Heer. 2019. Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations. Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (2019).
[51] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. https://www.nngroup.com/articles/response-times-3-important-limits/. Accessed: 2024-07-28.
[52] Nushoin. 2025. RTree - A C++ Implementation of R-Tree Data Structure. https://github.com/nushoin/RTree. Accessed: 2025-01-20.
[53] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and effective distribution-key recommendation for amazon redshift. Proceedings of the VLDB Endowment 13, 12 (2020), 2411–2423.
[54] Marianne Procopio, Ab Mosca, Carlos Scheidegger, Eugene Wu, and Remco Chang. 2021. Impact of cognitive biases on progressive visualization. IEEE Transactions on Visualization and Computer Graphics 28, 9 (2021), 3093–3112.
[55] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In Proceedings of the 2019 International Conference on Management of Data. 1981–1984.
[56] Nithin Reddy. 2023. Weather Data Insights. https://www.kaggle.com/code/nithinreddy90/weather-data-insights. Accessed: 2024-10-07.

[57] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. InfoVis (2015).
[58] Sloan Digital Sky Survey. 2024. Sloan Digital Sky Survey. https://www.sdss.org/. Accessed: 2024-08-13.
[59] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. IEEE Transactions on Visualization and Computer Graphics 8, 1 (2002), 52–65.
[60] Wenbo Tao, Xiaoyu Liu, Yedi Wang, Leilani Battle, Çağatay Demiralp, Remco Chang, and Michael Stonebraker. 2019. Kyrix: Interactive pan/zoom visualizations at scale. In Computer Graphics Forum, Vol. 38. Wiley Online Library, 529–540.
[61] Immanuel Trummer and Christoph Koch. 2016. Multi-objective parametric query optimization. ACM SIGMOD Record 45, 1 (2016), 24–31.
[62] Synthetic Data Vault. 2024. The Synthetic Data Vault. Put synthetic data to work! https://sdv.dev/
[63] Junran Yang, Hyekang Kevin Joo, Sai Yerramreddy, Dominik Moritz, and Leilani Battle. 2024. Optimizing Dataflow Systems for Scalable Interactive Visualization. Proceedings of the ACM on Management of Data 2, 1 (2024), 1–25.
[64] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 design advisor: Integrated automatic physical database design. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. 1087–1097.

Received October 2024; revised January 2025; accepted February 2025